Ornamentation in ML

Thomas Williams, Didier Rémy

Inria

Edinburgh, June 20, 2017
Similar types

\[
\text{type } \text{nat} = Z \mid S \quad \text{of} \quad \text{nat} \\
\text{type } \alpha \; \text{list} = \text{Nil} \mid \text{Cons} \; \text{of} \; \alpha \times \alpha \; \text{list}
\]

Similar values

\[
S (S (S (S (Z)))) \quad \text{Cons} (1, \text{Cons} (2, \text{Cons} (3, \text{Nil})))
\]

Ornament relation

\[
\text{type } \text{ornament } \alpha \; \text{natlist} : \text{nat} \rightarrow \alpha \; \text{list} \quad \text{with} \\
| \; Z \rightarrow \text{Nil} \\
| \; S \; \text{x}s \rightarrow \text{Cons} (_ , \; \text{x}s)
\]

The relation \( \alpha \; \text{natlist} \) between \text{nat} and \( \alpha \; \text{list} \) defines an \text{ornament}. 
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)

Coherence

add (length m) (length n) = length (append m n)
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)

Coherence
add (length m) (length n) = length (append m n)
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)

Coherence

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

let rec append m n = ?
Lifting

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

let rec append m n = ?

add (length m) (length n) = length (append m n)
**Lifting**

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

let rec append m n = ?
```

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

We restrict to **syntactic** lifting, following the structure of the original function.
let rec add m n = match m with
| Z → n
| S m' → S (add m' n)
let rec add m n = match m with
    | Z → n
    | S m’ → S (add m’ n)

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

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

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

Output

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

```
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)
```

Output

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

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)

Output

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

About nat & list

- Canonical, but trivial example
- Still, small enough to be a good running example, to explain the details of lifting.
More examples

About nat & list

- Canonical, but trivial example
- Still, small enough to be a good running example, to explain the details of lifting.

Many other use cases of lifting

- Pure refactoring: nothing to guess, no need for patches
- Special case: optimizing data representation
- Dealing with administrative stuff, e.g. locations:
  Write the code without locations and lift it to the code with locations
- Lifting a library, e.g. sets into maps
- Composing liftings: relifting lifted code
- Decomposing lifting into pure refactoring and a true, but simpler lifting
Pure refactoring

```plaintext
type exp =
  | Const of int
  | Add of exp × exp
  | Mul of exp × exp

type binop' = Add' | Mul'

type exp' =
  | Const' of int
  | Bin' of binop' × exp' × exp'
```
Pure refactoring

type exp =
| Const of int
| Add of exp × exp
| Mul of exp × exp

type binop' = Add' | Mul'

type exp' =
| Const' of int
| Bin' of binop' × exp' × exp'

type ornament oexp : exp → exp' with
| Const i → Const' i
| Add(u, v) → Bin'(Add', u, v)
| Mul(u, v) → Bin'(Mul', u, v)
Pure refactoring

type exp =
  | Const of int
  | Add of exp × exp
  | Mul of exp × exp
type binop’ = Add’ | Mul’
type exp’ =
  | Const’ of int
  | Bin’ of binop’ × exp’ × exp’

type ornament oexp : exp → exp’ with
  | Const i → Const’ i
  | Add(u, v) → Bin’(Add’, u, v)
  | Mul(u, v) → Bin’(Mul’, u, v)

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

let eval’ = lifting eval: oexp → int
Pure refactoring

**Type of Expression**

```ocaml
type exp =
  | Const of int
  | Add of exp × exp
  | Mul of exp × exp
```

**Type of Binary Operation’**

```ocaml
type binop’ = Add’ | Mul’
```

**Type of Modified Expression’**

```ocaml
type exp’ =
  | Const’ of int
  | Bin’ of binop’ × exp’ × exp’
```

**Type of Ornament**

```ocaml
type ornament oexp : exp → exp’ with
  | Const i → Const’ i
  | Add(u, v) → Bin’(Add’, u, v)
  | Mul(u, v) → Bin’(Mul’, u, v)
```

**Recursive Definition of Evaluation**

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

let eval’ = lifting eval : oexp → int
```

**Recursive Definition of Modified Evaluation**

```ocaml
let rec eval’ e = match e with
  | Const’ x → x
  | Bin’(Add’, x, x’) → add (eval’ x) (eval’ x’)
  | Bin’(Mul’, x, x’) → mul (eval’ x) (eval’ x’)
```

(4)7 / 27
Why not just rely on the typechecker?

- We do automatically what the programmer must do manually.
- We guarantee that the program obtained is related to the original one.
- The typechecker misses some places where a change is necessary.
Why not just rely on the typechecker?

- We do automatically what the programmer must do manually.
- We guarantee that the program obtained is related to the original one.
- The typechecker misses some places where a change is necessary.

Permuting values

```ml
type bool = False | True
```

We can safely exchange `True` and `False` in some places:
Why not just rely on the typechecker?

- We do automatically what the programmer must do manually.
- We guarantee that the program obtained is related to the original one.
- The typechecker misses some places where a change is necessary.

Permuting values

```plaintext
type bool = False | True
```

We can safely exchange True and False in some places:

```plaintext
type ornament not : bool → bool with
    | True → False
    | False → True
```

The relations between bare and ornamented values are tracked through the program (by ornament inference).
(Semi automated) code specialization

- Remove a field that is instantiated with `unit`
- Represent several boolean fields on a single integer
- Switch to a representation that can be unboxed (`bool option`)
Our goal

- Show that ornaments are a convenient tool for the ML programmer
- Design (the building blocks of) a language for meta-programming with ornamentation in ML
- Follow a composable approach, where ornamentation can be combined with other transformations, e.g. other forms of code inference, mixed with user interaction, etc.
- Lift ML programs to other ML programs
- Ensure that ornamentation is well-behaved
- Also an experiment in typed-based, user-driven code transformations.
our inspiration
Abstraction is our inspiration

Code reuse by abstraction *a priori* as a design principle, an easy case:

Polymorphic code or a function

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

Specialize the code with type and value arguments

\[ F \ A \]
Abstraction is our inspiration

Code reuse by abstraction *a priori* as a design principle, an easy case:

Polymorphic code or a function
\[ \Lambda(\alpha, \beta) \ldots \lambda(x : \tau, y : \sigma) \ M \]

Specialize the code with type and value arguments
\[ F \ A \]

Reuse the code with other type and value arguments
\[ F \ B \]
Abstraction is our inspiration

Code reuse by abstraction *a priori* as a design principle, an easy case:

Polymorphic code or a function
\[ \Lambda(\alpha, \beta) \ldots \lambda(x : \tau, y : \sigma) \ M \]

- Specialize the code with type and value arguments
  - \( F \ A \)
- Reuse the code with other type and value arguments
  - \( F \ B \)

Theorems for free
Parametricity ensures that the code \( F \ A \) and \( F \ B \) behaves the same up to the differences between \( A \) and \( B \).
Refactoring

base code

A
Refactoring

base code

Find its lifted version given an ornament specification

A

B
Refactoring by *abstraction a posteriori*

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

Inference

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

Find a generic version

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

**Inference**

\[
\begin{align*}
A &= A_{gen} \text{ id} \\
B &= A_{gen} \text{ args}
\end{align*}
\]
Refactoring by *abstraction a posteriori*

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

\[ A = A_{gen} \text{ id} \]

Find its lifted version given an ornament specification
\[ B \sim A_{gen} \text{ args} \]
Refactoring by *abstraction a posteriori*

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

Inference:
1. Infer args
2. Reduction
3. Simplification

Base code:
\[ A = A_{\text{gen}} \text{ id} \]

Find its lifted version given an ornament specification:
\[ B \sim A_{\text{gen}} \text{ args} \]
Refactoring by *abstraction a posteriori*

Find a generic version

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

1. Inference

2. Infer args

3. Meta-reduction

4. Simplification

Find its lifted version given an ornament specification

\[ A \sim B \]

\[ B \sim A_{gen \, args} \]

\[ A = A_{gen \, id} \]
Refactoring by *abstraction a posteriori*

Find a **most** generic version

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

**Inference**

1. infer args 2
2. meta-reduction 3
3. simplification 4

**Base code**

\[A = A_{gen} id\]

**Find its lifted version given an ornament specification**

\[B \sim A_{gen} args\]
Find a **most** generic version
\[ \Lambda(\alpha, \beta) \, \lambda(x : \tau)(y : \sigma) \, M \]

Find its lifted version given an ornament specification

\[ A \sim B \]

\[ A = A_{gen} \, id \]

\[ B \sim A_{gen} \, args \]
Questions & difficulties

The meta-language $mML$ must

- trace meta-reductions (easy by stratification)
- keep fine-grain invariants to ensure that it can be simplified to ML
- trace equalities between expressions for dead branches elimination
- have dependent types: type depends on pattern matching branches

The generic version

- depends solely on the source, not on the ornament (split of concerns)
- we restrict to the syntactic variants: we can only abstract over data-types that are explicit in the program (constructed or destructed)
- we abstract over all possible ornamentations of these data-types, respecting their recursive structure
Key Ideas from add to append...

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

- Insert conversions between lists and natS in add to obtain append.

```ocaml
let list2natS a = match a with
  | Nil → Z'
  | Cons(_,xs) → S' xs

let natS2list n x = match n with
  | Z' → Nil
  | S' xs → Cons(x, xs)
```
Key Ideas from add to append...

- Introduce a **skeleton** (open definition) of nat, to allows for hybrid nats where the head looks like a nat but the tail need not be a nat.
  
  ```ocaml
  type α natS = Z' | S' of α
  ```

- Insert conversions between lists and natS in add to obtain append.
  
  ```ocaml
  let list2natS a = match a with
  | Nil → Z'
  | Cons(_,xs) → S' xs

  let natS2list n x = match n with
  | Z' → Nil
  | S' xs → Cons(x, xs)

  let rec append m n =
  match list2natS m with
  | Z' → n
  | S' m' → natS2list (S' (append m’ n)) (List.hd m)
  ```
Key ideas . . . and to a generic lifting

From add to add_gen:

```ocaml
let append =
  let rec add m n =
    match list2natS m with
    | Z' → n
    | S' m' → natS2list (S' (add m' n)) (List.hd m)
  in add
```

abstract append over the ornament
Key ideas and to a generic lifting

From add to add_gen:

```
let add_gen =
  let rec add m n =
    match m2natS m with
    | Z → n
    | S' m' → natS2n (S' (add m' n)) (patch m n)
in add
```
Key ideas ... and to a generic lifting

From add to add_gen:

```ocaml
let add_gen m2natS =
  let rec add m n =
    match m2natS m with
    | Z’ → n
    | S’ m’ → natS2n (S’ (add m’ n)) (patch m n)
  in add
```

abstract append over the ornament

```ocaml
natS2n  patch =
```
Key ideas . . . and to a generic lifting

From add to add_gen:  

```ocaml
let add_gen m2natS natS2m n2natS natS2n patch =
  let rec add m n =
    match m2natS m with
    | Z' → n
    | S' m' → natS2n (S' (add m' n)) (patch m n)
in add
```

abstract append over the ornament
Key ideas ...and to a generic lifting

From add to add_gen:

\[
\text{let} \quad \text{add}_\text{gen} \quad \text{m2natS} \quad \text{natS2m} \quad \text{n2natS} \quad \text{natS2n} \quad \text{patch} = \\
\text{let rec} \quad \text{add} \; m \; n = \\
\quad \text{match} \; \text{m2natS} \; m \; \text{with} \\
\quad \mid \; \text{Z}' \; \rightarrow \; \text{n} \\
\quad \mid \; \text{S'} \; m' \; \rightarrow \; \text{natS2n} \; (\text{S'} \; (\text{add} \; m' \; n)) \; (\text{patch} \; m \; n) \\
\text{in} \quad \text{add}
\]

From add_gen back to append

\[
\text{let} \quad \text{append} = \text{add}_\text{gen} \quad \text{list2natS} \quad \text{natS2list} \quad \text{list2natS} \quad \text{natS2list} \\
\quad (\text{fun} \; m \; _ \; \rightarrow \; \text{match} \; m \; \text{with} \; \text{Cons}(x,_) \; \rightarrow \; x)
\]
Key ideas ... and to a generic lifting

From add to add_gen:

```ml
let add_gen m2natS natS2m n2natS natS2n patch =
  let rec add m n =
    match m2natS m with
    | Z' → n
    | S' m' → natS2n (S' (add m' n)) (patch m n)
  in add
```

abstract append over the ornament

From add_gen back to append

```ml
let append = add_gen list2natS natS2list list2natS natS2list
  (fun m _ → match m with Cons(x,_) → x)
```

From add_gen back to add: by passing the “identity” ornament

```ml
let nat2natS = function Z → Z' | S m → S' m
let natS2nat n x = match n with Z' → Z | S' m' → S m'
let add = add_gen nat2natS natS2nat nat2natS natS2nat
  (fun _ _ → ())
```
Staging

We need to

- 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
Staging

We need to

▸ 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:

```ocaml
let add_gen = fun m2natS natS2m n2natS natS2n patch →
  let rec add m n =
    match m2natS m with
    | Z' → n
    | S' m' → natS2n S' (add m' n) patch m n
  in add

let append = add_gen list2natS natS2list list2natS natS2list
  (fun m _ → match m with Cons(x,_) → x)
```
We need to

- 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:

```ocaml
let add_gen = fun m2natS natS2m n2natS natS2n patch =>
  let rec add m n =
    match m2natS # m with
      | Z' -> n
      | S' m' -> natS2n # S' (add m' n) # patch m n
  in add

let append = add_gen # list2natS # natS2list # list2natS # natS2list
  # (fun m _ -> match m with Cons(x,_) -> x)
```
Meta-reduction of the lifted code

```ocaml
let add_gen = fun m2natS natS2m n2natS natS2n patch ⇒
  let rec add m n =
    match m2natS m with
    | Z' ⇒ n
    | S' m' ⇒ natS2n S' (add m' n) # patch m n
  in add

let append = add_gen #list2natS# natS2list # list2natS #natS2list
  # (fun m _ ⇒ match m with Cons(x,_) ⇒ x)
```

- Reduce #-redexes at compile time.
- All #-abstractions and #-applications can actually be reduced.
- This ensured by typing!
Meta-reduction of the lifted code

```ml
let add_gen = fun m2natS natS2m n2natS natS2n patch ⇒
    let rec add m n =
        match m2natS m with
        | Z' → n
        | S' m' → natS2n (S' (add m' n)) # patch m n
    in add

let append = add_gen # list2natS # natS2list # list2natS # natS2list
             # (fun m _ → match m with Cons(x,_) → x)
```

- Reduce #-redexes at compile time.
- All #-abstractions and #-applications can actually be reduced.
- This ensured by typing!
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(List.hd m, zs))

- There remains some redundant pattern matchings...
- Decoding list to natS and encoding natS to list.
- We can eliminate the last one by reduction
let rec append m n =
match (match m with
  | Nil   → Z'
  | Cons(_, xs) → S’ xs)
with
  | Z’ → n
  | S’ m’ →
    Cons(List.hd m, append m’ n)
Eliminating the encoding

let rec append m n =
  match (match m with
    | Nil → Z'
    | Cons(_, xs) → S' xs) with
  | Z' → n
  | S' m' → Cons(List.hd m, append m' n)
Eliminating the encoding

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

And the other by extrusion... (commuting matches)
let rec append m n =
match m with
  | Nil  →
    (match Z' with
      | Z' → n
      | S' m' → Cons(List.hd m, append m' n))
  | Cons(_, xs) →
    (match S' xs' with
      | Z' → n
      | S' m' → Cons(List.hd m, append m' n))
Eliminating the encoding

```ocaml
let rec append m n =
  match m with
  | Nil → n
  | Cons(_, xs) →
    Cons(List.hd m, append m' n)
```

and reducing again
let rec append m n =
  match m with
  | Nil → n
  | Cons (x, xs)
    → Cons (List.hd m, append m' n)
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 eliminates all our uses of dependent types.
- This is always the case
In practice

We have a prototype implementation

- It follows the process outlined here.
- User interface issues: for specifying the instantiation, we take labelled patches and ornaments.
- To build the generic lifting, we transform deep pattern matching into shallow pattern matching.
- We try to recover the shape of the original program in a post-processing phase, keeping sharing annotations during duplication.
- We also expand local polymorphic lets (only a user interface problem)

See http://gallium.inria.fr/~remy/ornaments/

Goal: next version of the prototype for OCaml to run larger examples.
Discussion

Effects
- We use call-by-value, carefully preserving the evaluation order
- Should work without surprise in the presence of effects.
- A formal result about effects?

Recursion
- Modifying the recursive structure. Allowing mutual recursion.
- Non-regular types. GADTs.

Patches
- Can we write robust patches (that resist to code transformations)?
- Combine with some form of code inference (for patches)

Questions
- Should we give the user access to the intermediate language mML?
- Can we use mML for other purposes?
Take away

About ornaments

- Ornaments are useful in ML, both for software reuse and evolution.
- Going from the source program to the target program via a generic lifting that is later instantiated seems the right approach:
  - correctness by parametricity.
  - also allows to represent partially instantiated terms (user interface).
- We can even generate user-readable code!

Software evolution

- Ornaments are one way of doing software evolution.
- Software evolution via abstraction *a posteriori* seems a good principle, with other potential applications.
- Typed languages are a good setting for software evolution/refactoring that we should also explore further.
1. Dependent types
2. A meta-language for ornamentation
3. Encoding ornaments in mML
4. More examples
Outline

1. Dependent types
2. A meta-language for ornamentation
3. Encoding ornaments in mML
4. More examples
The case for dependent types

What if we add data to the Z constructor too?

```ocaml
type α stream = End | Continued | More of α × α stream

ornament α natstream : nat → α stream with
  | Z → (End | Continued)
  | S n → More (_, n)
```
What if we add data to the Z constructor too?

```ocaml
type α stream = End | Continued | More of α × α stream

ornament α natstream : nat → α stream with
  | Z → (End | Continued)
  | S n → More (_, n)

let natS2stream n x = match n with
  | Z' → (match x with
        | true → Continued
        | false → End)
  | S' n' → More (x, n')
```
The case for dependent types

What if we add data to the Z constructor too?

\[
\text{type } \alpha \text{ stream } = \text{End } | \text{ Continued } | \text{ More of } \alpha \times \alpha \text{ stream}
\]

\[
\text{ornament } \alpha \text{ natstream : nat } \rightarrow \alpha \text{ stream with}
| \text{Z } \rightarrow (\text{End } | \text{ Continued})
| \text{S } n \rightarrow \text{ More } (_, n)
\]

\[
\text{let natS2stream } n \ x = \text{match } n \text{ with}
| \text{Z’ } \rightarrow (\text{match } x \text{ with}
| \text{true } \rightarrow \text{ Continued}
| \text{false } \rightarrow \text{ End})
| \text{S’ } n’ \rightarrow \text{ More } (x, n’)
\]

What is the type of natS2stream?
The case for dependent types

What if we add data to the Z constructor too?

\[
\text{type } \alpha \text{ stream } = \text{End} \mid \text{Continued} \mid \text{More of } \alpha \times \alpha \text{ stream}
\]

\[
\text{ornament } \alpha \text{ natstream } : \text{nat } \to \alpha \text{ stream with}
\]

\[
|Z \to (\text{End } \mid \text{Continued})
\]

\[
|S \ n \to \text{More } (\_, \ n)
\]

\[
\text{let natS2stream } n \ x = \text{match } n \text{ with}
\]

\[
|Z' \to (\text{match } x \text{ with}
\]

\[
|\text{true } \to \text{Continued}
\]

\[
|\text{false } \to \text{End})
\]

\[
|S' \ n' \to \text{More } (x, \ n')
\]

What is the type of \(x\)?

\[
(\text{match } x \text{ with } Z' \to \text{unit} \mid S' \ _ \to \alpha)
\]
The case for dependent types

What if we add data to the $Z$ constructor too?

```agda
type α stream = End | Continued | More of α × α stream

ornament α natstream : nat → α stream with
  | Z → (End | Continued)
  | S n → More (_ , n)

let natS2stream n x = match n with
  | Z' → (match x with
       | true → Continued
       | false → End)
  | S' n' → More (x , n')
```

What is the type of natS2stream?

$$\lambda^#\alpha. \Pi(x : \text{natS (list } \alpha)).$$
$$\Pi(y : (\text{match } x \text{ with } Z' \rightarrow \text{unit} | S' _ \rightarrow \alpha)). \text{ list } \alpha$$
What if we add data to the Z constructor too?

\[
\text{type } \alpha \text{ stream} = \text{End } | \text{Continued } | \text{More of } \alpha \times \alpha \text{ stream}
\]

\[
\text{ornament } \alpha \text{ natstream} : \text{nat } \rightarrow \alpha \text{ stream with}
\]
\[
| \text{Z } \rightarrow (\text{End } | \text{Continued})
\]
\[
| \text{S n } \rightarrow \text{More } (_) \times n)
\]

\[
\text{let natS2stream n x = match n with}
\]
\[
| \text{Z' } \rightarrow (\text{match x with}
\]
\[
| \text{true } \rightarrow \text{Continued}
\]
\[
| \text{false } \rightarrow \text{End})
\]
\[
| \text{S' n' } \rightarrow \text{More } (x, n')
\]

What is the type of natS2stream?

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

Dependent types we introduce can always be eliminated.
The case for dependent types

The type may depend on more than the constructor.

```
| type α list01 =
|   | Nil01
|   | Cons0 of α list01
|   | Cons1 of α × α list01

| ornament α olist01 : bool list → α list01 with
|   | Nil → Nil01
|   | Cons (False, xs) → Cons0 (xs)
|   | Cons (True, xs) → Cons1 (_ , xs)

| match m with
|   | Nil' → unit
|   | Cons' (False, _) → unit
|   | Cons' (True, _) → α
```
Outline

1. Dependent types
2. A meta-language for ornamentation
3. Encoding ornaments in mML
4. More examples
Starting from ML

\[\begin{align*}
\tau, \sigma &::= \alpha \mid \tau \rightarrow \tau \mid \zeta \overline{\tau} \mid \forall (\alpha : \text{Typ}) \tau \\
a, b &::= x \mid \text{let } x = a \text{ in } a \mid \text{fix } (x : \tau) x. a \mid a a \\
&\quad \mid \Lambda (\alpha : \text{Typ}). u \mid a \tau \mid d \overline{\tau} \overline{a} \mid \text{match } a \text{ with } \overline{P} \rightarrow a \\
P &::= d \overline{\tau} \overline{x}
\end{align*}\]
Starting from ML

\[
E ::= [] \mid E \ a \mid \nu \ E \mid d(\bar{v}, E, \bar{a}) \mid \Lambda(\alpha : \text{Typ}). \ E \mid E \ \tau \\
| \quad \text{match } E \text{ with } \overline{P} \rightarrow a \mid \text{let } x = E \text{ in } a
\]

\[
(fix (x : \tau) \ y. \ a) \ \nu \quad \overset{\beta}{\longrightarrow} \quad a[x \leftarrow \text{fix} (x : \tau) \ y. \ a, y \leftarrow \nu]
\]

\[
(\Lambda(\alpha : \text{Typ}). \ \nu) \ \tau \quad \overset{\beta}{\longrightarrow} \quad \nu[\alpha \leftarrow \tau]
\]

\[
\text{let } x = \nu \text{ in } a \quad \overset{\beta}{\longrightarrow} \quad a[x \leftarrow \nu]
\]

\[
\text{match } d_j \overline{\tau_j} (v_i)^i \text{ with } (d_j \overline{\tau_j} (x_{ji})^i \rightarrow a_j)^j \quad \overset{\beta}{\longrightarrow} \quad a_j[x_{ij} \leftarrow v_i]^i
\]

Context-Beta

\[
\begin{align*}
a & \overset{\beta}{\longrightarrow} b \\
\hline
E[a] & \overset{\beta}{\longrightarrow} E[b]
\end{align*}
\]
From ML to \( m\text{ML} \)

- eML: add type-level pattern matching and equalities.
- \( m\text{ML} \): add dependent, meta-abstraction and application.

Reduction (under some typing conditions):
- From \( m\text{ML} \), reduce meta-application and get a term in eML
- From eML, eliminate type-level pattern matching and get a term in ML
eML is obtained by extending the type system of ML.
eML is obtained by extending the type system of ML.

\[ \Gamma = \alpha : \text{Typ}, m : \text{nat}' \ (\text{list } \alpha), x : \text{match } m \text{ with } Z' \rightarrow \text{unit} \mid S' \ _ \rightarrow \alpha \]

Consider:

\[
\text{match } m \text{ with } \\
\mid Z' \rightarrow \text{Nil} \\
\mid S' \ m' \rightarrow \text{Cons} (x, m')
\]
eML is obtained by extending the type system of ML.

\[ \Gamma = \alpha : \text{Typ}, m : \text{nat'} (\text{list } \alpha), x : \text{match } m \text{ with } \text{Z'} \rightarrow \text{unit} \mid \text{S'} \_ \rightarrow \alpha \]

Consider:

\[
\text{match } m \text{ with} \\
\quad \mid \text{Z'} \rightarrow \text{Nil} \\
\quad \mid \text{S'} m' \rightarrow \text{Cons } (x, m')
\]

In the \text{S'} branch, we know \( m = \text{S'} m' \).

Thus:

\[
x : \text{match } m \text{ with } \text{Z'} \rightarrow \text{unit} \mid \text{S'} \_ \rightarrow \alpha \\
\quad = \text{match } S' m' \text{ with } \text{Z'} \rightarrow \text{unit} \mid \text{S'} \_ \rightarrow \alpha \\
\quad = \alpha
\]
Equalities

We extend the typing environment with equalities:

\[ \Gamma ::= \ldots \mid \Gamma, a =_\tau b \]

Introduced on pattern matching

\[
\Gamma \vdash \tau : \text{Sch} \quad (d_i : \forall (\alpha_k : \text{Typ})^k (\tau_{ij})^j \rightarrow \zeta (\alpha_k)^k)^i \quad \Gamma \vdash a : \zeta (\tau_k)^k \\
(\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, a =_{\zeta(\tau_k)^k} d_i(\tau_{ij})^k(x_{ij})^j \vdash b_i : \tau)^i \\
\hline \\
\Gamma \vdash \text{match } a \text{ with } (d_i(\tau_{ik})^k(x_{ij})^j \rightarrow b_i)^i : \tau
\]

Used to prove type equalities

Since terms appear in types, they generate equalities on types, which allows for implicit conversions:

\[
\Gamma \vdash \tau_1 \equiv \tau_2 \quad \Gamma \vdash a : \tau_1 \\
\hline \\
\Gamma \vdash a : \tau_2
\]
Elimination of equalities

We restrict reduction in equalities so that it remains decidable.

Assume a is term an eML a such that $\Gamma \vdash a : \tau$, where $\Gamma$ and $\tau$ are in ML. Then, we can transform $a$ into a well-typed ML term by:

- Using an equalities to substitute in terms
- Extruding nested pattern matching
- Reducing pattern matching

This justifies the use of eML as an intermediate language for ornamentation
Meta-programming in \textit{mML}

We introduce a separate type for meta-functions, so that they can only be applied using meta-application.

\[
(\lambda^\#(x:\tau). a) \# u \rightarrow^\# a[x \leftarrow u]
\]

This enables to eliminate all abstractions and applications marked with \#.

We restrict types so that meta-constructions can not be manipulated by the ML fragment.
If there are no meta-typed variables in the context, the meta-reduction $\rightarrow^#$ will eliminate all meta constructions and give an eML term.

But the meta-reduction also commutes with the ML reduction.

We thus have two dynamic semantics for the same term:

- For reasoning, we can consider that meta and ML reduction are interleaved.
- We can use the meta reduction in the first stage to compile an mML term down to an eML term.
Dependent functions

We need dependent types for the encoding function:

\[
\text{natS2list} : \lambda \# \alpha. \Pi(x : \text{natS} (\text{list } \alpha)). \\
\Pi(y : \text{match } x \text{ with } Z' \to \text{unit} | S' \_ \to \alpha). \\
\text{list } \alpha
\]
Dependent functions

We need dependent types for the encoding function:

\[
\text{natS2list} : \lambda^\sharp \alpha. \Pi(x : \text{natS} (\text{list} \ \alpha)).
\]
\[
\Pi(y : \text{match} \ x \ \text{with} \ Z' \to \text{unit} \ \mid \ S' \_ \to \alpha).
\]
\[
\text{list} \ \alpha
\]

For the encoding of ornaments to type correctly, we also add:

- Type-level functions to represent the type of the extra information.
- The ability to abstract on equalities so they can be passed to patches.
Outline

1. Dependent types
2. A meta-language for ornamentation
3. Encoding ornaments in mML
4. More examples
let append = lifting add : \( \alpha \) natlist \( \rightarrow \) \( \alpha \) natlist \( \rightarrow \) \( \alpha \) natlist

We mean:

- If \( ml \) is a lifting of \( m \) (for natlist)
- and \( nl \) is a lifting of \( n \) (for natlist)
- then \( append \ ml \ nl \) is a lifting of \( add \ m \ n \) (for natlist)
let append = lifting add : α natlist → α natlist → α natlist

We mean:

- If \( ml \) is a lifting of \( m \) (for natlist)
- and \( nl \) is a lifting of \( n \) (for natlist)
- then \( append \ ml \ nl \) is a lifting of \( add \ m \ n \) (for natlist)

We build a (step-indexed) binary logical relation on \( mML \), and add an interpretation for datatype ornaments.

The interpretation of a functional lifting is exactly the interpretation of function types, replacing “is a lifting of” by “is related to”.
Datatype ornaments

A datatype ornament naturally gives a relation:

\[
\text{ornament } \alpha \text{ natlist : nat } \rightarrow \alpha \text{ list with }
\]

\[
| \ Z \rightarrow \text{Nil} \\
| \ S \ xs \rightarrow \text{Cons}(_, \ xs)
\]
A datatype ornament naturally gives a relation:

\[
\text{ornament } \alpha \text{ natlist : nat } \rightarrow \alpha \text{ list with }
\]

\[
\begin{align*}
| & Z \rightarrow \text{Nil} \\
| & S \times s \rightarrow \text{Cons}(_{}, s) \\
\end{align*}
\]

\[(Z, \text{Nil}) \in \mathcal{V}[\text{natlist } \tau] \quad \frac{(u, v) \in \mathcal{V}[\text{natlist } \tau]}{(S \ u, \text{Cons} (a, v)) \in \mathcal{V}[\text{natlist } \tau]}\]
Datatype ornaments

A datatype ornament naturally gives a relation:

\[ \text{ornament } \alpha \text{ natlist} : \text{nat} \rightarrow \alpha \text{ list with} \]
\[ | \ Z \rightarrow \text{Nil} \]
\[ | \ S \times x \rightarrow \text{Cons(}_a, \times x) \]

\[(Z, \text{Nil}) \in \mathcal{V}[\text{natlist } \tau] \]
\[ (u, v) \in \mathcal{V}[\text{natlist } \tau] \]
\[ (S \ u, \text{Cons}(a, v)) \in \mathcal{V}[\text{natlist } \tau] \]

We prove that the ornamentation functions are correct relatively to this definition:

- if we construct a natural number and a list from the same skeleton, they are related;
- if we destruct related values, we obtain the same skeleton.
Correctness

- Consider a term $a_-$.
- Generalize it into $a$. By the fundamental lemma, $a$ is related to itself.
- Construct an instanciation $\gamma_+$ and the identity instanciation $\gamma_-$. 
  - $\gamma_-(a)$ and $\gamma_+(a)$ are related.
  - $\gamma_-(a)$ reduces to $a_-$, preserving the relation.
- Simplify $\gamma_+(a)$ into $a_+$ (an ML term), preserving the relation.
- $a_-$ and $a_+$ are related.
1. Dependent types

2. A meta-language for ornamentation

3. Encoding ornaments in mML

4. More examples
Specialization: unit map

type $\alpha$ map =
  | Node of $\alpha$ map $\times$ key $\times$ $\alpha$ $\times$ $\alpha$ map
  | Leaf
Specialization: unit map

```haskell
type α map =
  | Node of α map × key × α × α map
  | Leaf

Instead of unit map, we could use a more compact representation:

type set =
  | SNode of set × key × set
  | SLeaf

type ornament mapset : unit map → set with
  | Node(l,k,(),r) → SNode(l,k,r)
  | Leaf → SLeaf
```
Specialization: unboxing

type \( \alpha \) option =
- None
- Some of \( \alpha \)

type booloption =
- NoneBool
- SomeTrue
- SomeFalse
Specialization: unboxing

type \(\alpha\) option =
| None 
| Some of \(\alpha\)

\textbf{type} booloption =
| NoneBool 
| SomeTrue 
| SomeFalse

\textbf{type ornament} boolopt : bool option \to booloption with
| None \to NoneBool 
| Some(true) \to SomeTrue 
| Some(false) \to SomeFalse