Ornaments in Practice

Thomas Williams
Encadré par: Pierre-Évariste Dagand, Didier Rémy

INRIA - Gallium

September 8, 2014

Motivation

Two very similar functions

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

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

Motivation

Two very similar functions

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

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

Coherent

`add (length ml) (length nl) = length (append ml nl)`

Naturals and lists

Similar types

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

S    (    S    (    S    (    Z  )))
Cons(1, Cons(2, Cons(3, Nil)))
```

Projection function

```
let rec length = function
  | Nil → Z
  | Cons(x, xs) → S(length xs)
```

Motivation

Trees

```
type tree =
  | LLeaf
  | LNode of ltree × ltree
```

```
type α ntree =
  | NLeaf
  | NNode of α ntree × α
              × α ntree
```

GADTs

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

```
type (_,α) vec =
  | VNil : (zero, α) vec
  | VCons : α × (n, α) vec
      → (n succ, α) vec
```

```
type zero = Zero   type _ succ = Succ
```

Ornaments (McBride,2010; Dagand,2012)

Ornaments in ML

Ornaments were developed in type theory. Can they be adapted to ML?

Toy implementation: a preprocessor for a small System F-like language with GADTs. It adapts easily to ML: we will assume this in the examples.

Contents

A syntax for ornaments

An ornament is defined by a *projection function* from the ornamented type to the bare type.

```
let rec length = function
  | Nil → Z
  | Cons(x, xs) → S(length xs)
```

The function is subject to some syntactic restrictions to ensure it preserves the recursive structure. They are checked by the system when declaring an ornament:

```
ornament from length : α list → nat
```

Lifting functions

Coherence

```
length (append ml nl) = add (length ml) (length nl)
```

Lifting functions

Coherence

```
length (append ml nl) = add (length ml) (length nl)

project (f_lifted x y) = f (project x) (project y)
```

The function `f_lifted` is a lifting of `f`.

Syntactic lifting

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

Syntactic lifting

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

```
let lifting append from add with
  {length} → {length} → {length}
```

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}
```

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}
```

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

Syntactic lifting

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

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

let lifting append from add with
  {length} → {length} → {length}

let rec append ml nl =
```

Syntactic lifting

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

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

```
let lifting append from add with
  {length} → {length} → {length}
```

```
let rec append ml nl =
```

$$\boxed{\texttt{m} \xleftarrow{\text{length}} \texttt{ml}}$$

$$\boxed{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}$$

Syntactic lifting

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

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

```
let lifting append from add with
  {length} → {length} → {length}
```

```
let rec append ml nl = match      with
```

$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$

$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$

Syntactic lifting

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

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

let lifting append from add with
  {length} → {length} → {length}

let rec append ml nl = match     with
```

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

let rec append ml nl = match ml with
```

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$

$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$

Syntactic lifting

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

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

```
let lifting append from add with
  {length} → {length} → {length}
```

```
let rec append ml nl = match ml with
```

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

let rec append ml nl = match ml with
```

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

$$\boxed{\texttt{Z} \xleftarrow{\text{length}} \texttt{Nil}}$$

$$\boxed{\frac{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{S(n)} \xleftarrow{\text{length}} \texttt{Cons(x,nl)}}}$$

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

let rec append ml nl = match ml with
  | Nil →
```

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

$$\texttt{Z} \xleftarrow{\text{length}} \texttt{Nil}$$

$$\frac{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{S(n)} \xleftarrow{\text{length}} \texttt{Cons(x,nl)}}$$

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

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

$$\frac{\mathtt{m'} \xleftarrow{\text{length}} \mathtt{ml'} \qquad \mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}}{\mathtt{add\ m'\ n} \xleftarrow{\text{length}} \mathtt{append\ ml'\ nl}}$$

$$\mathtt{Z} \xleftarrow{\text{length}} \mathtt{Nil}$$

$$\frac{\mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}}{\mathtt{S(n)} \xleftarrow{\text{length}} \mathtt{Cons(x,nl)}}$$

$$\mathtt{m} \xleftarrow{\text{length}} \mathtt{ml}$$

$$\mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}$$

Syntactic lifting

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

```
let lifting append from add with
  {length} → {length} → {length}
```

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

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

$$\texttt{Z} \xleftarrow{\text{length}} \texttt{Nil}$$

$$\boxed{\frac{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{S(n)} \xleftarrow{\text{length}} \texttt{Cons(x,nl)}}}$$

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$$

$$\boxed{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'}}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

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

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

$$\texttt{Z} \xleftarrow{\text{length}} \texttt{Nil}$$

$$\frac{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{S(n)} \xleftarrow{\text{length}} \texttt{Cons(x,nl)}}$$

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\boxed{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}$$

$$\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

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

$$\frac{\mathtt{m'} \xleftarrow{\text{length}} \mathtt{ml'} \qquad \mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}}{\mathtt{add\ m'\ n} \xleftarrow{\text{length}} \mathtt{append\ ml'\ nl}}$$

$$\mathtt{Z} \xleftarrow{\text{length}} \mathtt{Nil}$$

$$\frac{\mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}}{\mathtt{S(n)} \xleftarrow{\text{length}} \mathtt{Cons(x,nl)}}$$

$$\mathtt{m} \xleftarrow{\text{length}} \mathtt{ml}$$

$$\mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}$$

$$\mathtt{m'} \xleftarrow{\text{length}} \mathtt{ml'}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}


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

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

$$\texttt{Z} \xleftarrow{\text{length}} \texttt{Nil}$$

$$\frac{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{S(n)} \xleftarrow{\text{length}} \texttt{Cons(x,nl)}}$$

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$$

$$\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

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

$$\frac{\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'} \qquad \texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{add m' n} \xleftarrow{\text{length}} \texttt{append ml' nl}}$$

$$\texttt{Z} \xleftarrow{\text{length}} \texttt{Nil}$$

$$\frac{\texttt{n} \xleftarrow{\text{length}} \texttt{nl}}{\texttt{S(n)} \xleftarrow{\text{length}} \texttt{Cons(x,nl)}}$$

$$\texttt{m} \xleftarrow{\text{length}} \texttt{ml}$$

$$\texttt{n} \xleftarrow{\text{length}} \texttt{nl}$$

$$\texttt{m'} \xleftarrow{\text{length}} \texttt{ml'}$$

Syntactic lifting

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

let lifting append from add with
  {length} → {length} → {length}

let rec append ml nl = match ml with
  | Nil → nl
  | Cons(x,ml') →
      Cons( ? , append ml' nl)
```

$$\frac{\mathtt{m'} \xleftarrow{\text{length}} \mathtt{ml'} \qquad \mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}}{\mathtt{add\ m'\ n} \xleftarrow{\text{length}} \mathtt{append\ ml'\ nl}}$$

$$\mathtt{Z} \xleftarrow{\text{length}} \mathtt{Nil}$$

$$\frac{\mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}}{\mathtt{S(n)} \xleftarrow{\text{length}} \mathtt{Cons(x,nl)}}$$

$$\mathtt{m} \xleftarrow{\text{length}} \mathtt{ml}$$

$$\mathtt{n} \xleftarrow{\text{length}} \mathtt{nl}$$

$$\mathtt{m'} \xleftarrow{\text{length}} \mathtt{ml'}$$

Two phases

Syntactic lifting

```
let rec append ml nl = match ml with
  | Nil → nl
  | Cons(x,ml') → Cons( ? , append ml' nl)
```

The creative part

- Manually, by intervention of the programmer
- With a *patch* specifying what should be added where

```
let append from add
  with {length} → {length} → {length}
  patch fun _ → match _ with Cons(x, _) → Cons({x}, _)
```

- Code inference: `x` makes the most sense here

Coherence is not enough

```
let rec rev_append ml nl = match ml with
  | Nil → nl
  | Cons(x,ml') → rev_append ml' (Cons(x,nl))
```

```
length (rev_append ml nl) = add (length ml) (length nl)
```

The coherence condition is not strong enough to guide the automatic lifting process.

Contents

Ornaments for refactoring

```
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'
```

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

Ornaments for refactoring (2)

```
let rec conv : expr' → expr = function
  | Const'(i) → Const(i)
  | BinOp(Add', u, v) → Add(conv u, conv v)
  | BinOp(Mul', u, v) → Mul(conv u, conv v)
ornament from conv : expr' → expr

let lifting eval' from eval with {conv} → _
```

```
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
```

The lifting is unique, because conv is bijective.

Other applications

Large-scale lifting of data structures

- Ocaml's Set library to maps (sets with values)
- Including higher-order functions
- Only the values need to be propagated

GADTs

- A GADT is an ornament: constraints are added
- Lifting unique: the contents are the same
- In practice, works for what the typechecker could have proved

Contents

Syntactic ornament, binary term

```
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    )
```

Syntactic ornament, binary term

```
let rec     append   ml   nl = match   ml with
  |   Nil →   nl
  |   Cons({x},   ml') →   Cons({x},     append     ml'   nl)
```

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

Syntactic ornament, binary term

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

let rec add&append m&ml n&nl = match m&ml with
  | Z&Nil → n&nl
  | S&Cons({x},m’&ml’) → S&Cons({x},add&append m’&ml’ n&nl)

let rec append ml nl = match ml with
  | Nil → nl
  | Cons(x,ml’) → Cons(x,append ml’ nl)
```

Typing of ornaments

The & in names is only a notation, it has no meaning in the binary language.

Ornaments translate to binary type definitions.

```
type {α} nat&list =
  | Z&Nil
  | S&Cons of {α} ×{α} nat&list
```

The typing enforces ornamentation:

```
val add&append
  : {α}. {α} nat&list →{α} nat&list →{α} nat&list
```

The braces guarantee that values don't escape from the ornamented code to the bare code.

Lifting with binary terms

Lifting: finding a binary term that *projects* to the base term and has the right type.

1. The binary typing relation guarantees that we have a valid lifting.
2. The projections of a well-typed term are well-typed.
3. The complexity is preserved: the additional complexity comes only from the code added between brackets.
4. If the ornamented code terminates, the base code terminates too.

What's missing?

Semantic ornaments

We can recover a semantic definition using contextual equivalence. All syntactic rules remain admissible (it is *compatible*), it is a superset of syntactic equivalence (it is *adequate*), and equivalent to the definition using coherence.

Higher-order and nested ornaments

We can use this to understand what is a higher-order ornament and a nested ornament.

Conclusion

What we have learned

- Describing ornaments by projection is a good fit for ML
- Several classes of useful ornaments
- The syntactic lifting gives good, predictable results
- And can be well-explained by theory

Future work

- Better patches
- Integrating into ML: effects? inference?
- Combining ornaments: adding the invariants of two GADTs?

Questions ?

Ocaml integration

- Interaction with type inference: inferring the ornament specification of let-bound values?

```
let rev xs =
  let rec rev_append acc = function
    | x::xs → rev_append (x::acc) xs
    | [] → acc
  in
  rev_append [] xs
```

- Lifting effectful libraries?

Lifting more complex data structures

Sets

```
type t
val compare : t → t → int
type set = Empty | Node of t × set × set
```

Maps

```
type α map =
  | MEmpty
  | MNode of t × α × α map × α map
```

Ornament

```
let rec keys = function
  | MEmpty → Empty
  | MNode(k, v, l, r) → Node(k, keys l, keys r)
ornament from keys : α map → set
```

Lifting a higher-order function

```
let rec exists (p : t → bool) (s : set) : bool =
  match s with
    | Empty → false
    | Node(l, k, r) → p k || exists p l || exists p r
```

```
let lifting map_exists from exists
  with (t → +α → t) → {keys} → bool
```

Lifting a higher-order function

```
let rec exists (p : t → bool) (s : set) : bool =
  match s with
    | Empty → false
    | Node(l, k, r) → p k || exists p l || exists p r
```

```
let lifting map_exists from exists
  with (t → +α → t) → {keys} → bool
```

```
let rec map_exists p m =
  match m with
    | Empty → false
    | Node(l, k, v, r) → p k ? || map_exists p l
          || map_exists p r
```

GADTs

Several data structures with the same contents but different invariants, *i.e.* a constraint on the shape of the type.

Lists and vectors

```
type α list = Nil | Cons of α × α list
type zero = Zero            type _ succ = Succ
type (_, α) vec =
 | VNil : (zero, α) vec
 | VCons : α × (n, α) vec → (n succ, α) vec

let rec to_list : type n. (n, α) vec → α list =
  function
  | VNil → Nil
  | VCons(x, xs) → Cons(x, xs)
ornament from to_list : (γ, α) vec → α list
```

The lifting should be unambiguous.

Lifting for GADTs

Automatic for some invariants, we only need to give the expected type of the function:

```
let rec zip xs ys = match xs, ys with
  | Nil, Nil → Nil
  | Cons(x, xs), Cons(y, ys) → Cons((x, y), zip xs ys)
  | _ → failwith "different length"

let lifting vzip :
    type n. (n, α) vec → (n, β) vec → (n, α × β) vec
    from zip with {to_list} → {to_list} → {to_list}
```

Lifting for GADTs

Automatic for some invariants, we only need to give the expected type of the function:

```
let rec zip xs ys = match xs, ys with
  | Nil, Nil → Nil
  | Cons(x, xs), Cons(y, ys) → Cons((x, y), zip xs ys)
  | _ → failwith "different length"

let lifting vzip :
  type n. (n, α) vec → (n, β) vec → (n, α × β) vec
  from zip with {to_list} → {to_list} → {to_list}
```

```
let rec vzip :
  type n. (n, α) vec → (n, β) vec → (n, α × β) vec
  = fun xs ys → match xs, ys with
   | VNil, VNil → VNil
   | VCons(x, xs), VCons(y, ys) →
      VCons((x, y), vzip xs ys)
   | _ → failwith "different length"
```

When lifting fails

```
type (_, _, _) min =
  | MinS : (α, β, γ) min → (α su, β su, γ su) min
  | MinZl : (ze, α, ze) min
  | MinZr : (α, ze, ze) min

let lifting vzipm :
  type n1 n2 nmin.
    (n1, n2, nmin) min →
        (n1, α) vec → (n2, β) vec → (nmin, α × β) vec
  from zipm
  with +_ → {to_list} → {to_list} → {to_list}

let rec vzipm :
  type n1 n2 nmin. (n1, n2, nmin) min
   → (n1, α) vec → (n2, β) vec → (nmin, α × β) vec
  = fun m xs ys → match xs, ys with
  | VNil, VNil → VNil
  | VCons(x, xs), VCons(y, ys) →
     VCons((x, y), vzipm [ ? ] xs ys)
  | _, _ → failwith "different length"
```