Reasoning about GADT Pattern Matching in Haskell

George Karachalias

UNIVERSITEIT GENT

Partners

Supervised by

Tom Schrijvers (KU Leuven)

Joint work with

Dimitrios Vytiniotis (MSR Cambridge)

Simon Peyton Jones (MSR Cambridge)

Nikolaos Papaspyrou (NTUA)

Background

Algebraic Data Types

```
data List a
  = Nil
  | Cons a (List a)
```

Algebraic Data Types

```
data List a
   = Nil
   | Cons a (List a)


l :: List Int
l = Cons 1 (Cons 2 (Cons 3 Nil))
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
ghci> foo (Cons 1 Nil) (Cons 3 Nil)
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
ghci> foo (Cons 1 Nil) (Cons 3 Nil)
1
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
ghci> foo (Cons 1 Nil) (Cons 3 Nil)
1
```

```
ghci> foo (Cons 1 (Cons 2 Nil)) Nil
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

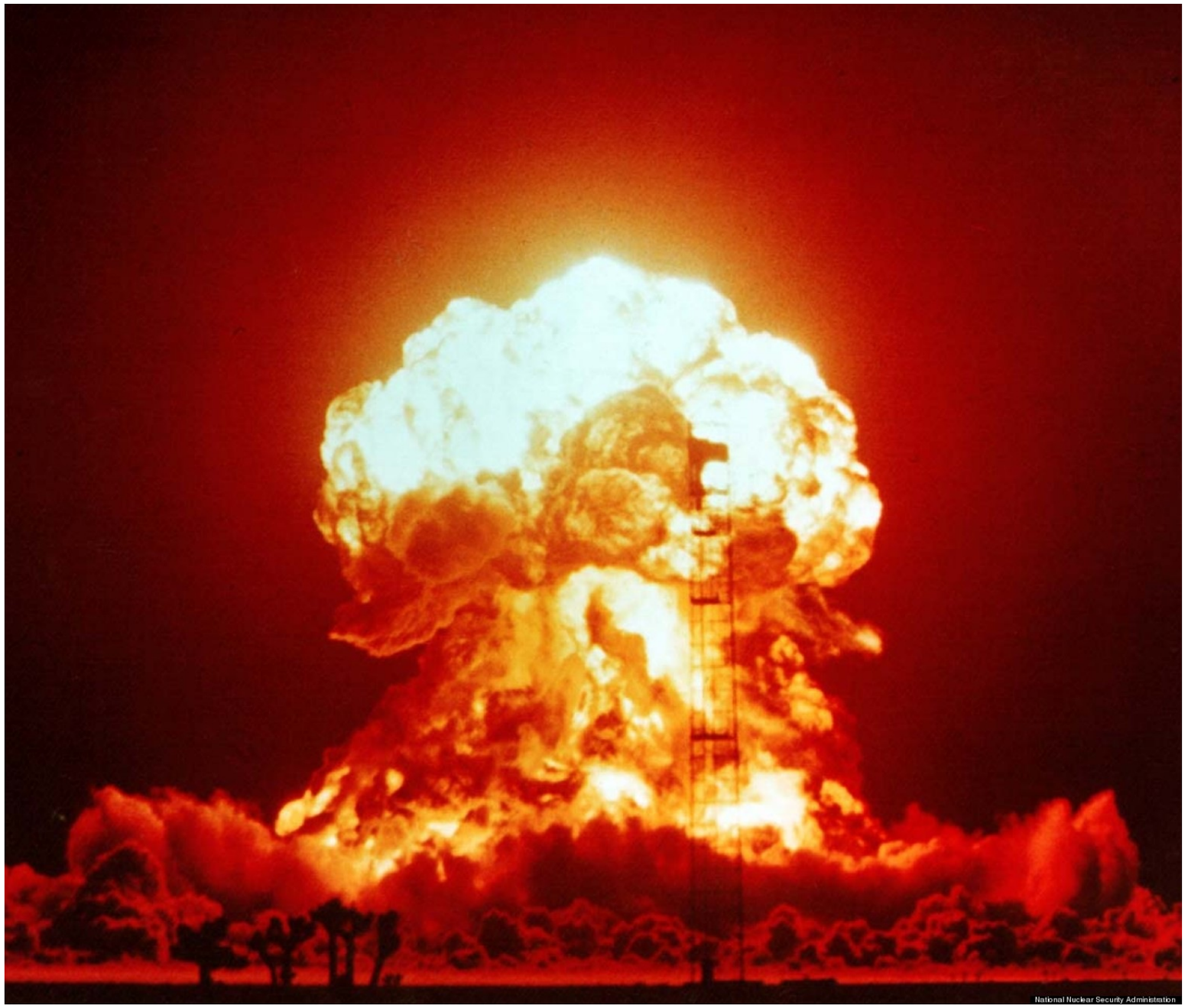
```
ghci> foo (Cons 1 Nil) (Cons 3 Nil)
1

ghci> foo (Cons 1 (Cons 2 Nil)) Nil
2
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
ghci> foo Nil (Cons 1 (Cons 2 Nil))
```

Non-Exhaustiveness



Exhaustiveness Checking

```
:set -fwarn-incomplete-patterns
```

Exhaustiveness Checking

```
:set -fwarn-incomplete-patterns
```

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

Exhaustiveness Checking

```
:set -fwarn-incomplete-patterns
```

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
Patterns not matched:
    Nil (Cons _ _)
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
ghci> foo Nil Nil
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
ghci> foo Nil Nil
2
```

Example

```
foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         =
```

```
ghci> foo Nil Nil
2
```

Redundancy Checking

```
:set -fwarn-overlapping-patterns
```

Redundancy Checking

```
:set -fwarn-overlapping-patterns

foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

Redundancy Checking

```
:set -fwarn-overlapping-patterns

foo :: List a -> List b -> Int
foo (Cons x xs) (Cons y ys) = 1
foo _           Nil         = 2
foo Nil         Nil         = 3
```

```
Pattern matches are overlapped:
    foo Nil Nil = 3
```

Summary

- Everything is rosy thanks to GHC's exhaustiveness/overlapping checker
- Based on Maranget's first technique [Mara94] for compiling lazy pattern matching

[Mara94] Luc Maranget, *"Two Techniques for Compiling Lazy Pattern Matching"*, Rapport de recherche RR-2385, INRIA, 1994. Projet PARA.

Generalized Algebraic Data Types (since 2006)

Not so rosy
anymore

GADTs

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  Cons :: a -> Vec n a -> Vec (Succ n) a

kind Nat = Zero | Succ Nat
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
```

```
Patterns not matched:
    Nil (Cons _ _)
    (Cons _ _) Nil
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
```

```
Patterns not matched:
    Nil (Cons _ _)
    (Cons _ _) Nil
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil         Nil         = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
```

```
Patterns not matched:
    Nil (Cons _ _)
    (Cons _ _) Nil
```

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  Cons :: a -> Vec n a -> Vec (Succ n) a
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil         Nil         = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
```

```
Patterns not matched:
    Nil (Cons _ _)
    (Cons _ _) Nil
```

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  Cons :: a -> Vec n a -> Vec (Succ n) a
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil         Nil         = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
```

```
Patterns not matched:
    Nil (Cons _ _)
    (Cons _ _) Nil
```

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  Cons :: a -> Vec n a -> Vec (Succ n) a
```

Attempted Fix

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip Nil          (Cons y ys) = error “Nil (Cons _ _)”
```

Attempted Fix

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip Nil          (Cons y ys) = error “Nil (Cons _ _)”
```

```
Error:
  Couldn’t match type ‘Zero’ with type ’Succ n1’
```

Attempted Fix

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip Nil          (Cons y ys) = error “Nil (Cons _ _)”
```

```
Error:
  Couldn’t match type ‘Zero’ with type ’Succ n1’
```

[Vyti11] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers and Martin Sulzmann, *“OutsideIn(X) modular type inference with local assumptions”*, J. Funct. Program., vol. 21, no. 4-5, pp. 333–412, September 2011.

Silencing the Compiler

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

The result?

- No trust in the emitted warnings
- Bogus overlapped patterns to suppress them

My Way

Example

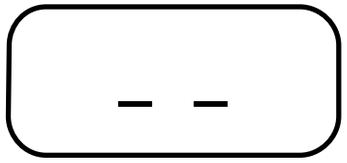
```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

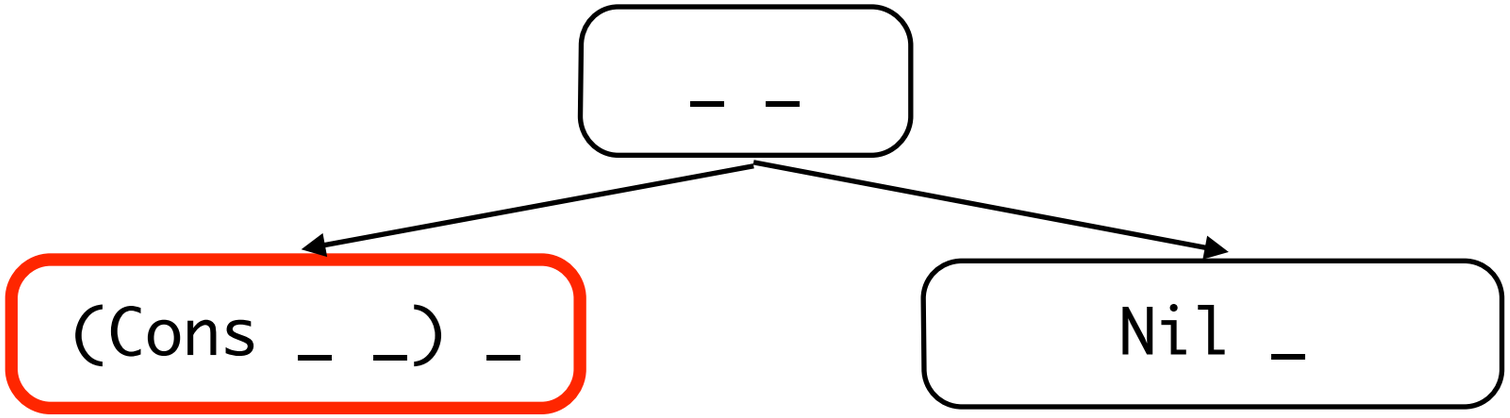
Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil         Nil         = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _           _           = error "type error”
```



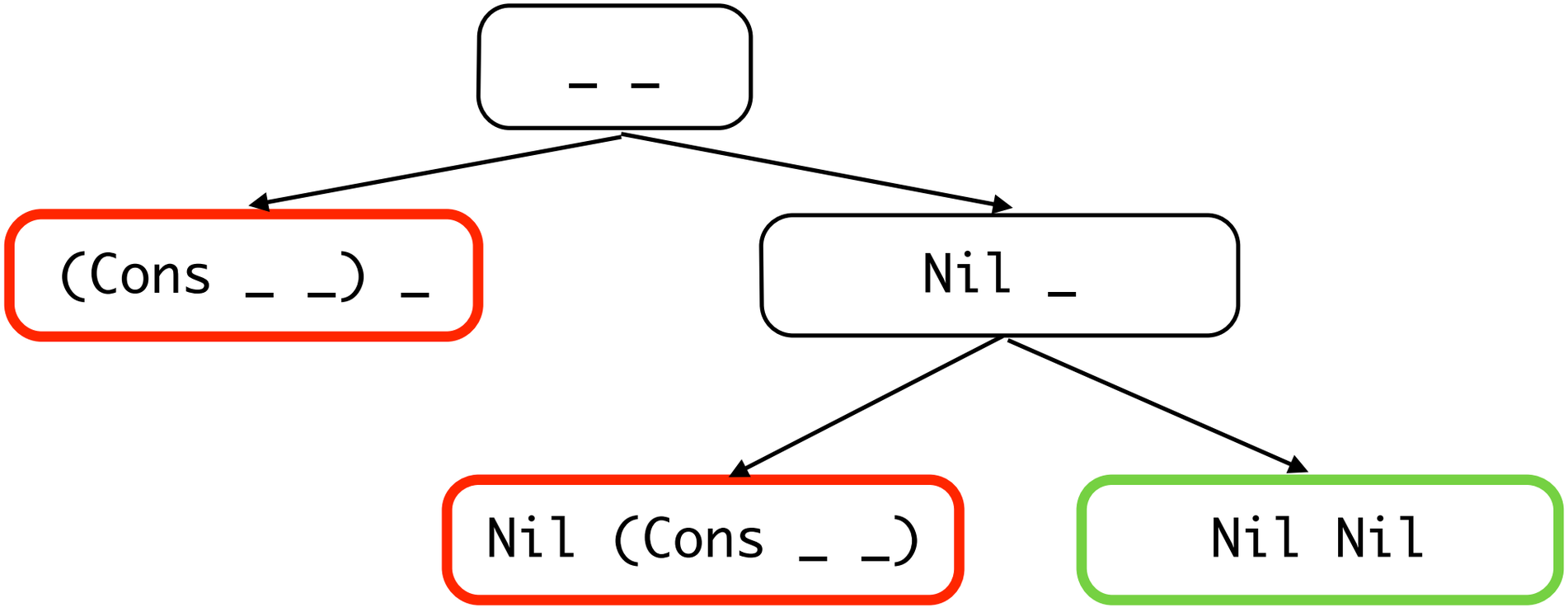
Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

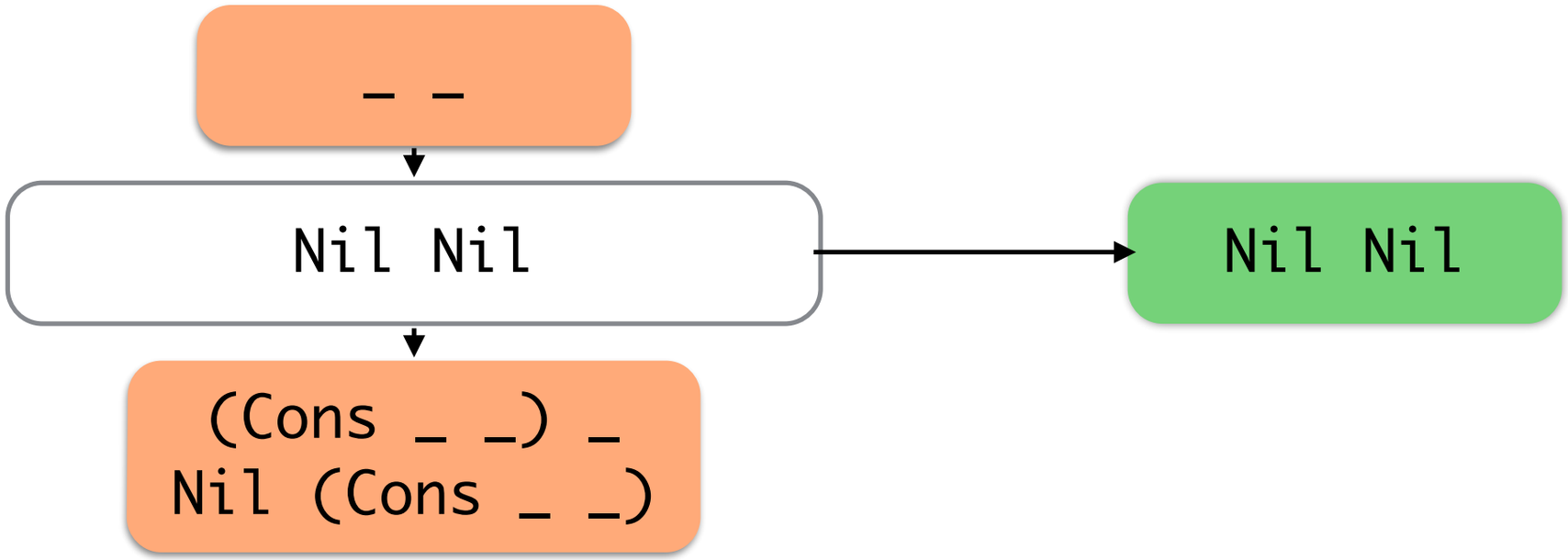


Example

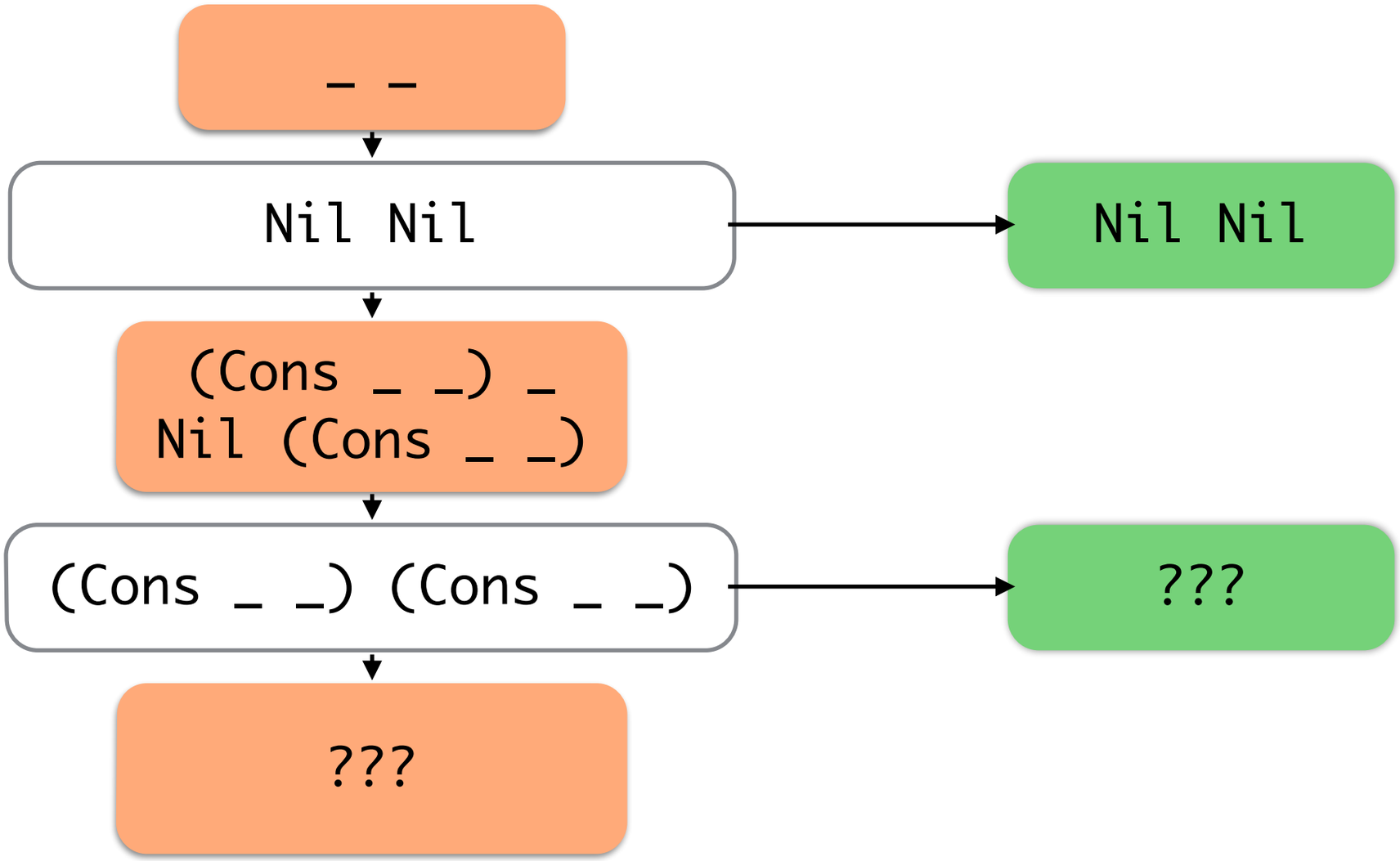
```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil         Nil         = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _           _           = error "type error”
```



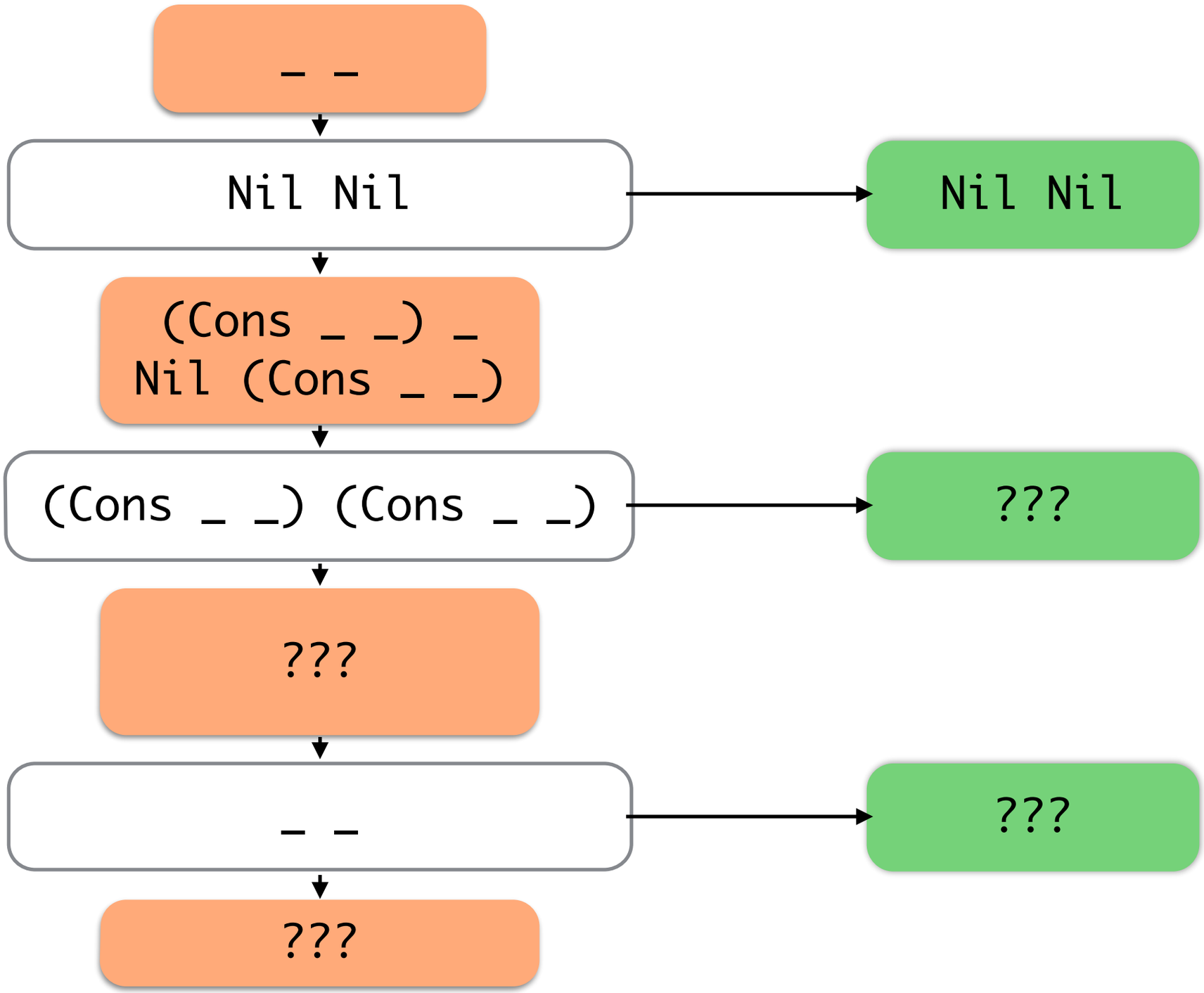
Example



Example



Example



Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

```
(Cons _ _) _
Nil (Cons _ _)
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

```
(Cons _ _) _
Nil (Cons _ _)
```

```
Nil (Cons _ _)
```

```
(Cons _ _) _
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _           = error "type error”
```

```
(Cons _ _) _
Nil (Cons _ _)
```

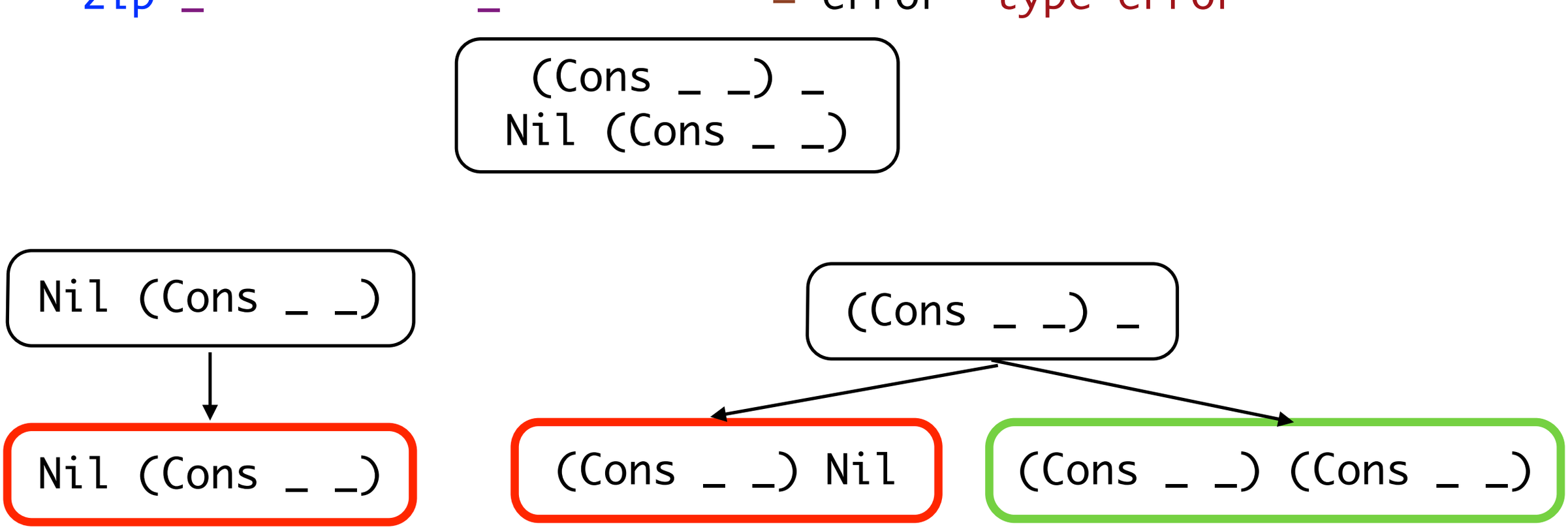
```
Nil (Cons _ _)
```

```
Nil (Cons _ _)
```

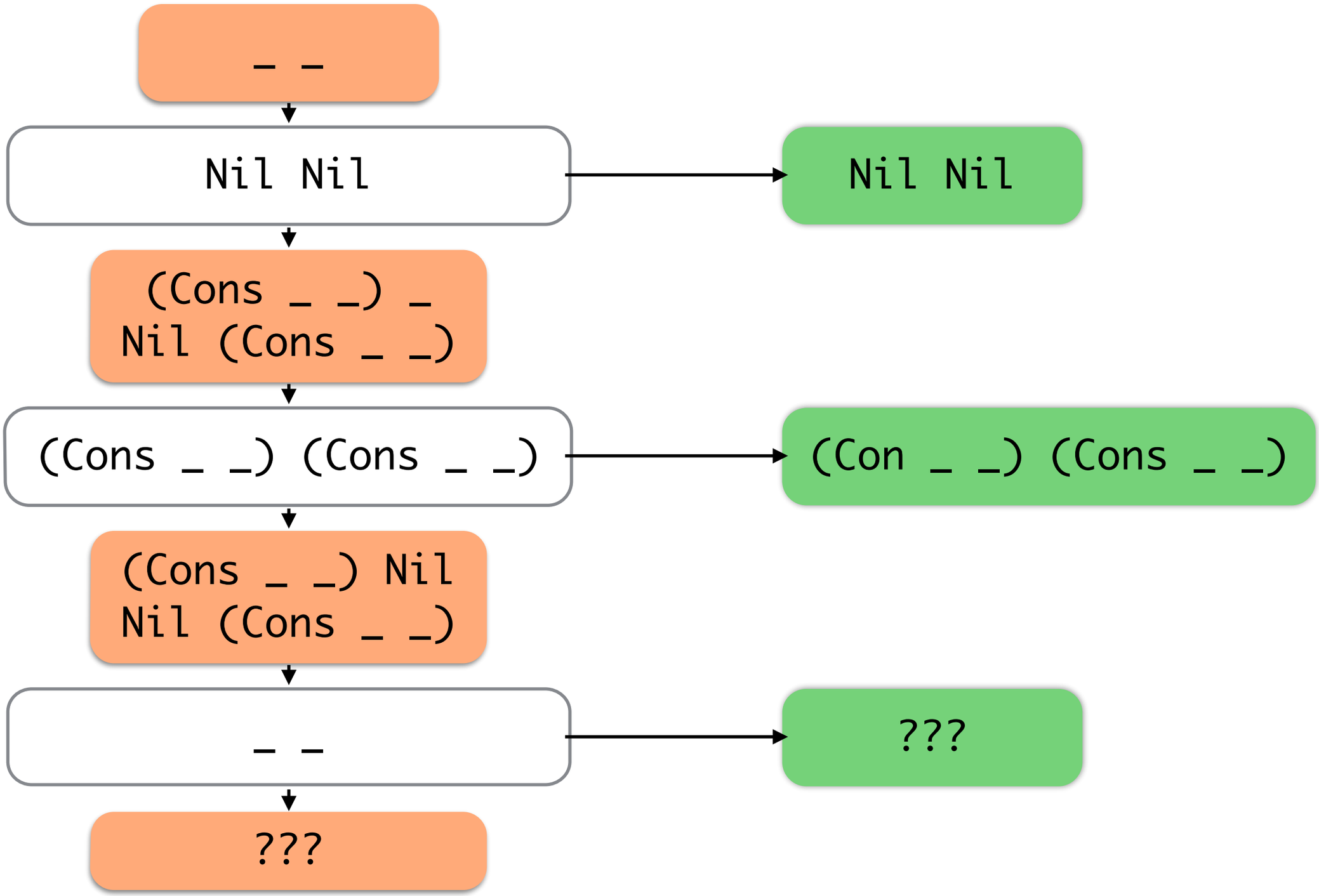
```
(Cons _ _) _
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```



Example


_ _
Nil Nil
Nil Nil
(Cons _ _) _
Nil (Cons _ _)
(Cons _ _) (Cons _ _)
(Con _ _) (Cons _ _)
(Cons _ _) Nil
Nil (Cons _ _)
_ _
???
???

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _           = error "type error”
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

```
Nil (Cons _ _)
(Cons _ _) Nil
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

```
Nil (Cons _ _)
(Cons _ _) Nil
```

```
Nil (Cons _ _)
```

```
(Cons _ _) Nil
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil         Nil         = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _           _           = error "type error”
```

```
Nil (Cons _ _)
(Cons _ _) Nil
```

```
Nil (Cons _ _)
```

```
(Cons _ _) Nil
```

```
Nil (Cons _ _)
```

Example

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
zip Nil          Nil          = Nil
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip _            _            = error "type error”
```

```
Nil (Cons _ _)
(Cons _ _) Nil
```

```
Nil (Cons _ _)
```

↓

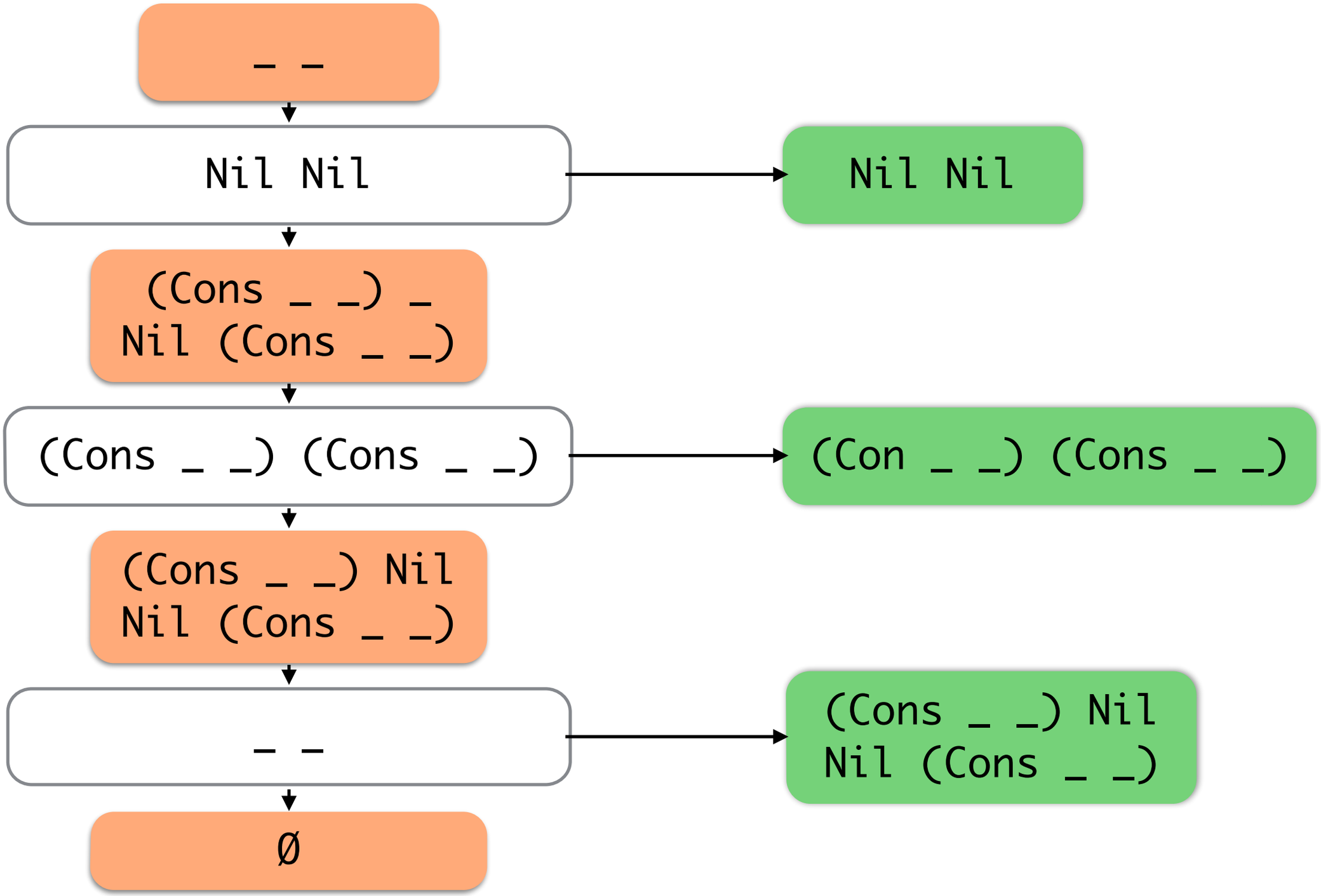
```
Nil (Cons _ _)
```

```
(Cons _ _) Nil
```

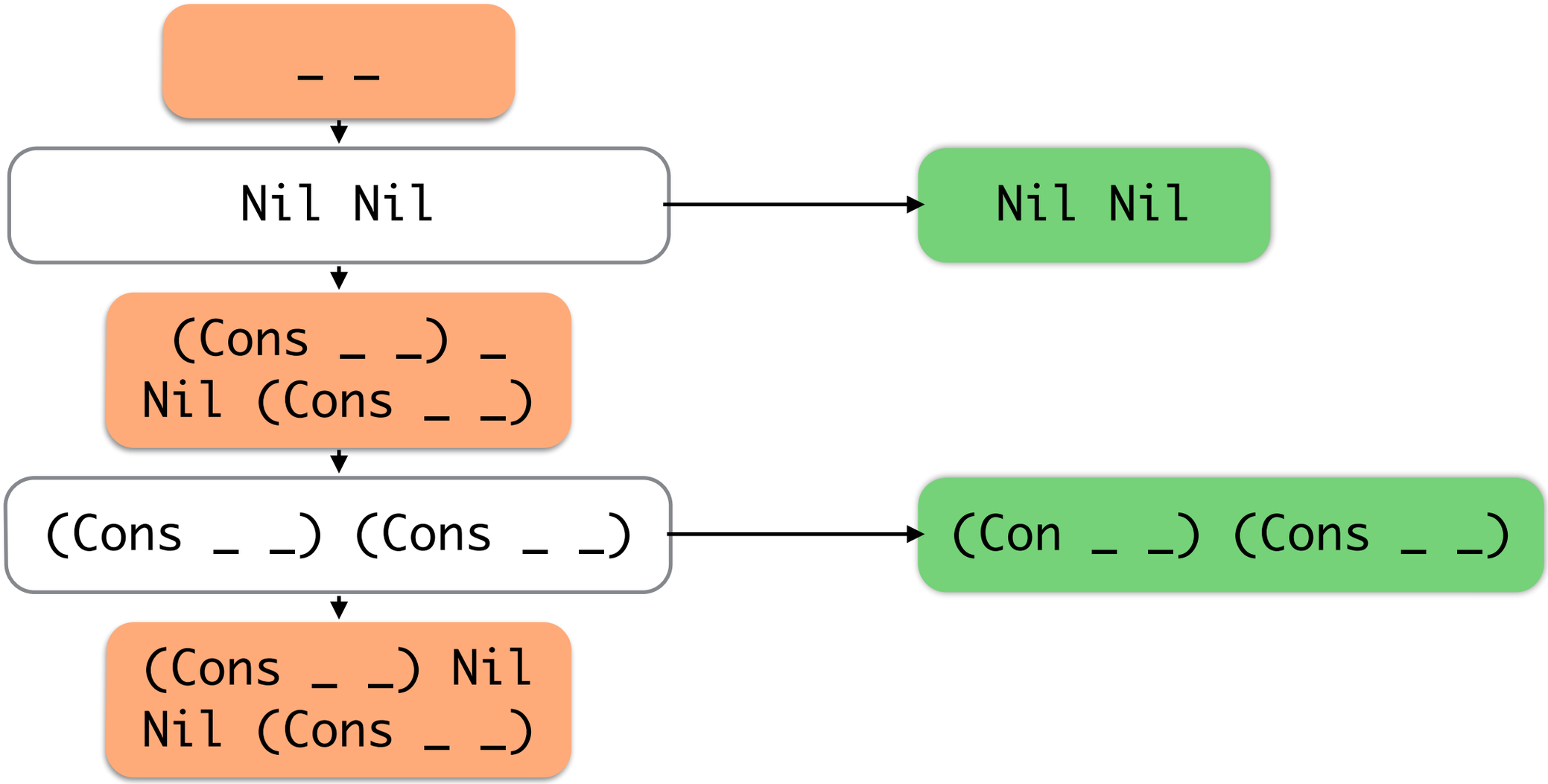
↓

```
(Cons _ _) Nil
```

Example


_ _
Nil Nil
Nil Nil
(Cons _ _) _
Nil (Cons _ _)
(Cons _ _) (Cons _ _)
(Con _ _) (Cons _ _)
(Cons _ _) Nil
Nil (Cons _ _)
_ _
(Cons _ _) Nil
Nil (Cons _ _)
Ø

Example


_ _
Nil Nil
Nil Nil
(Cons _ _) _
Nil (Cons _ _)
(Cons _ _) (Cons _ _)
(Con _ _) (Cons _ _)
(Cons _ _) Nil
Nil (Cons _ _)

Totality & Usefulness

Exhaustiveness Check

- A match is total if the final set of ***well-typed*** uncovered cases is empty.

Redundancy Check

- A clause is useful if the set of ***well-typed*** cases it covers is non-empty.

Type-checking

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
```

```
Nil Nil
```

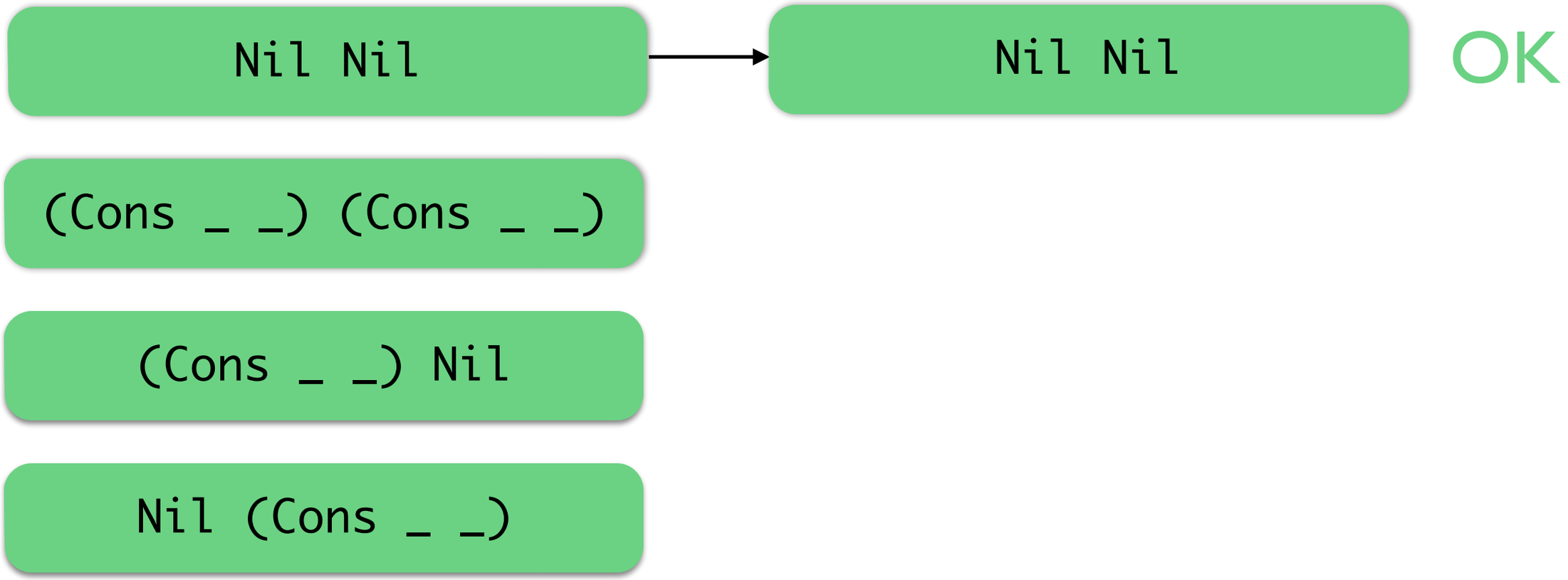
```
(Cons _ _) (Cons _ _)
```

```
(Cons _ _) Nil
```

```
Nil (Cons _ _)
```

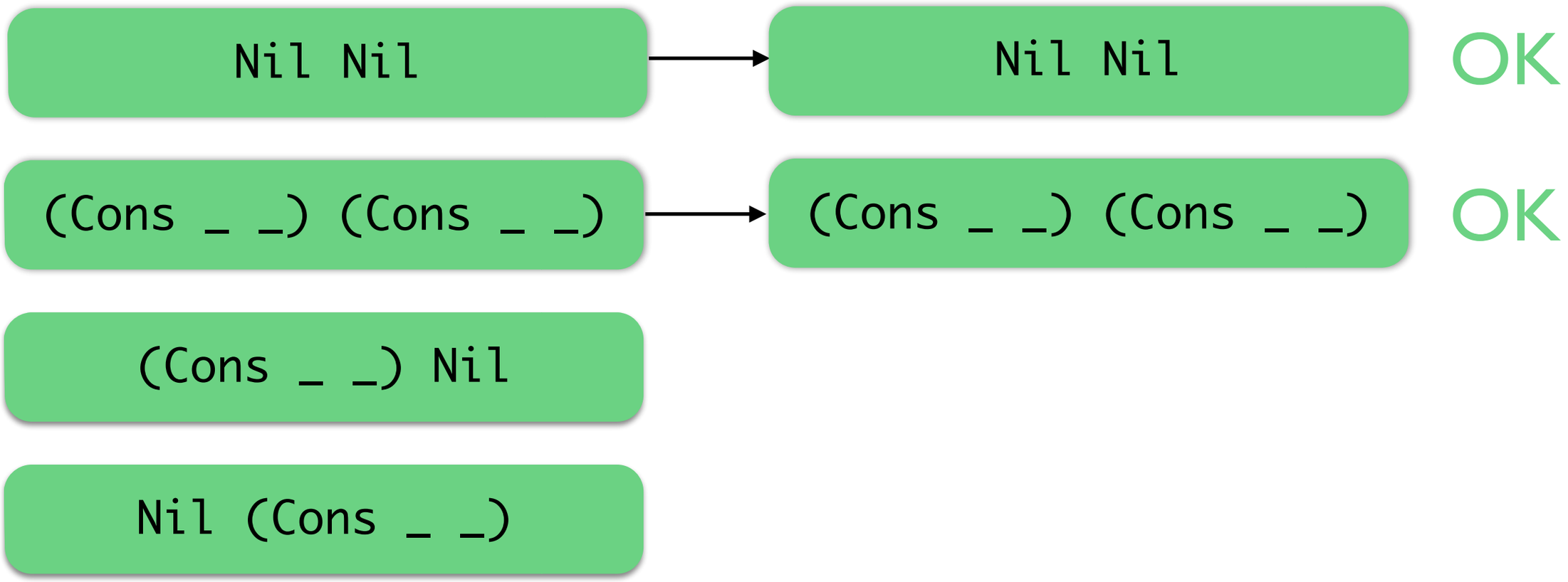
Type-checking

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
```



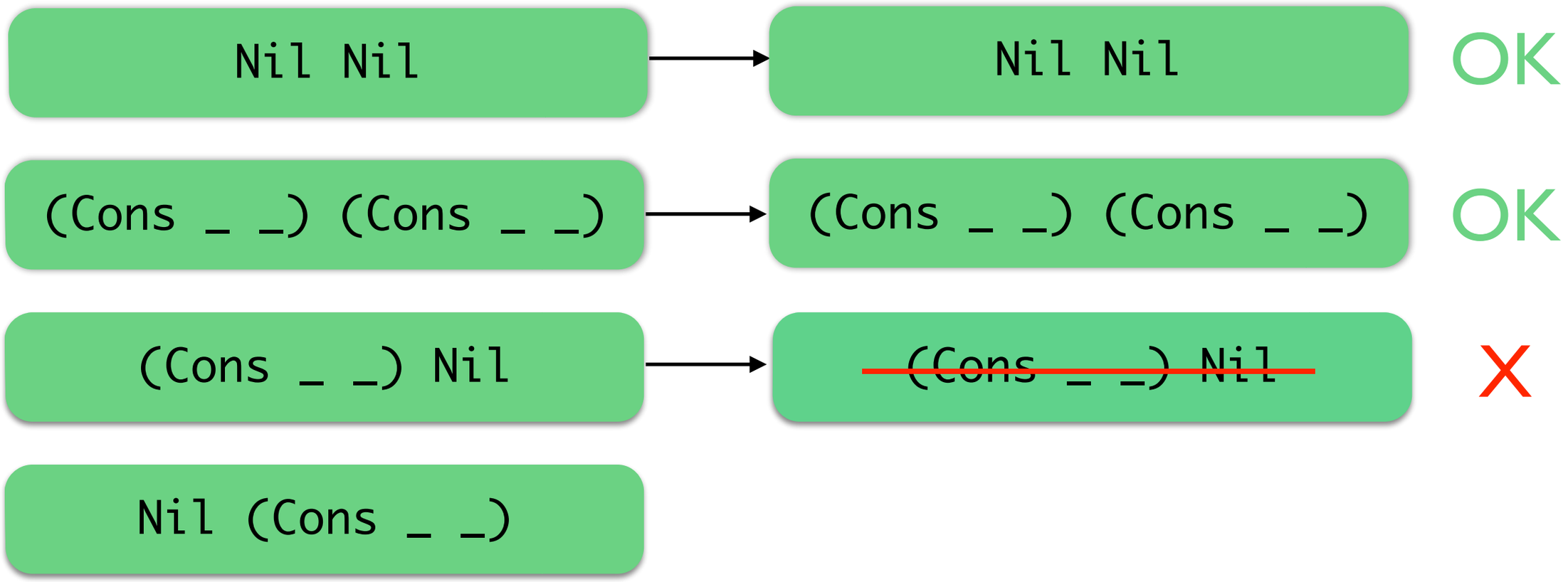
Type-checking

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
```



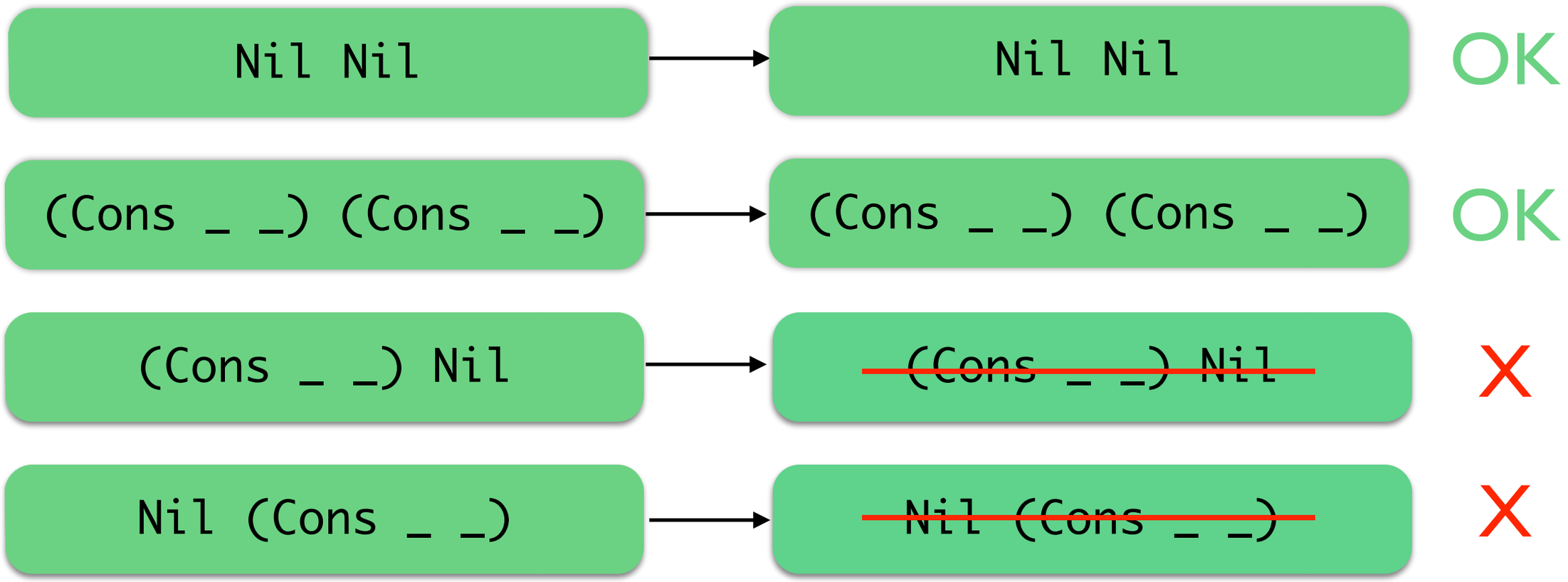
Type-checking

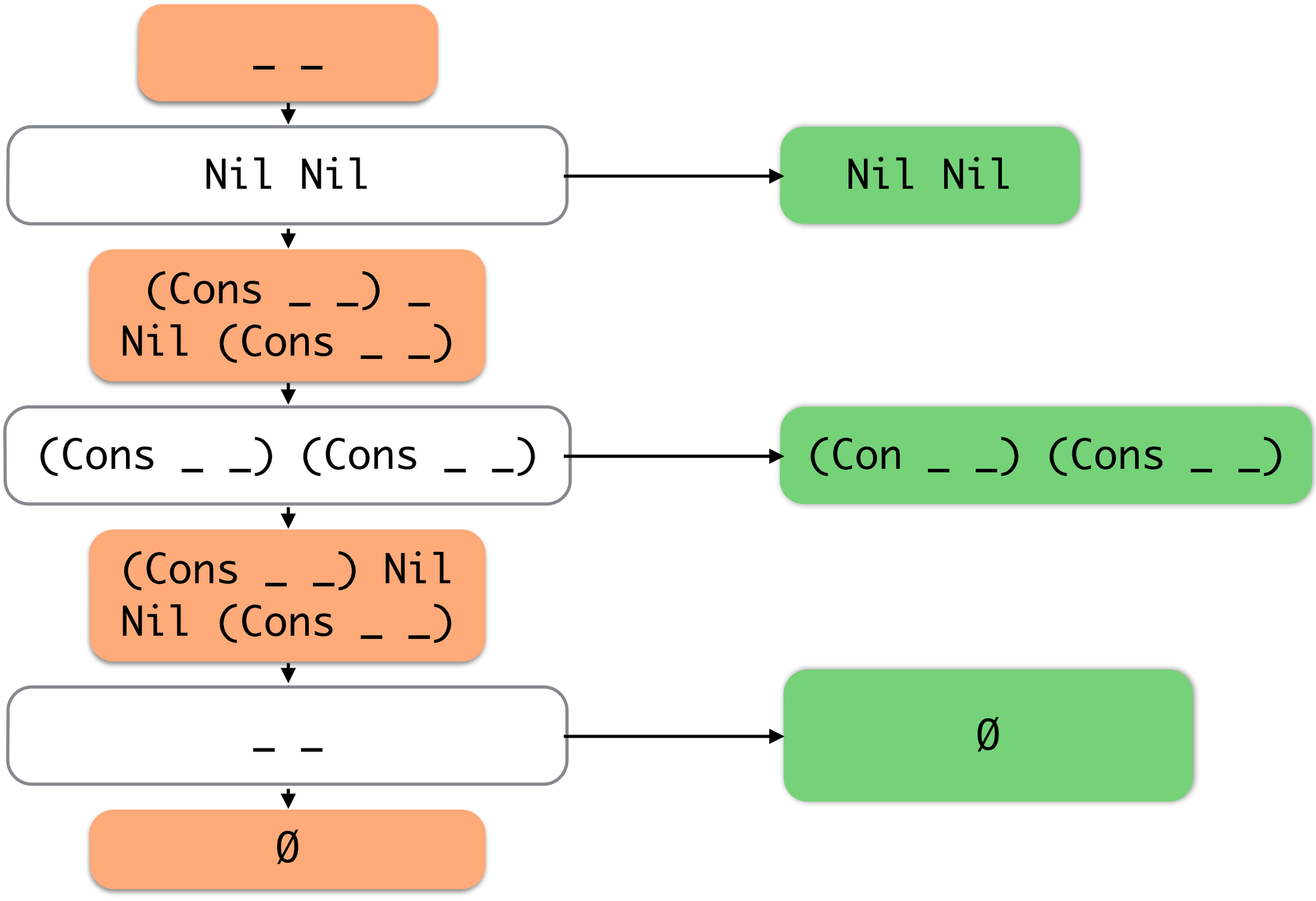
```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
```



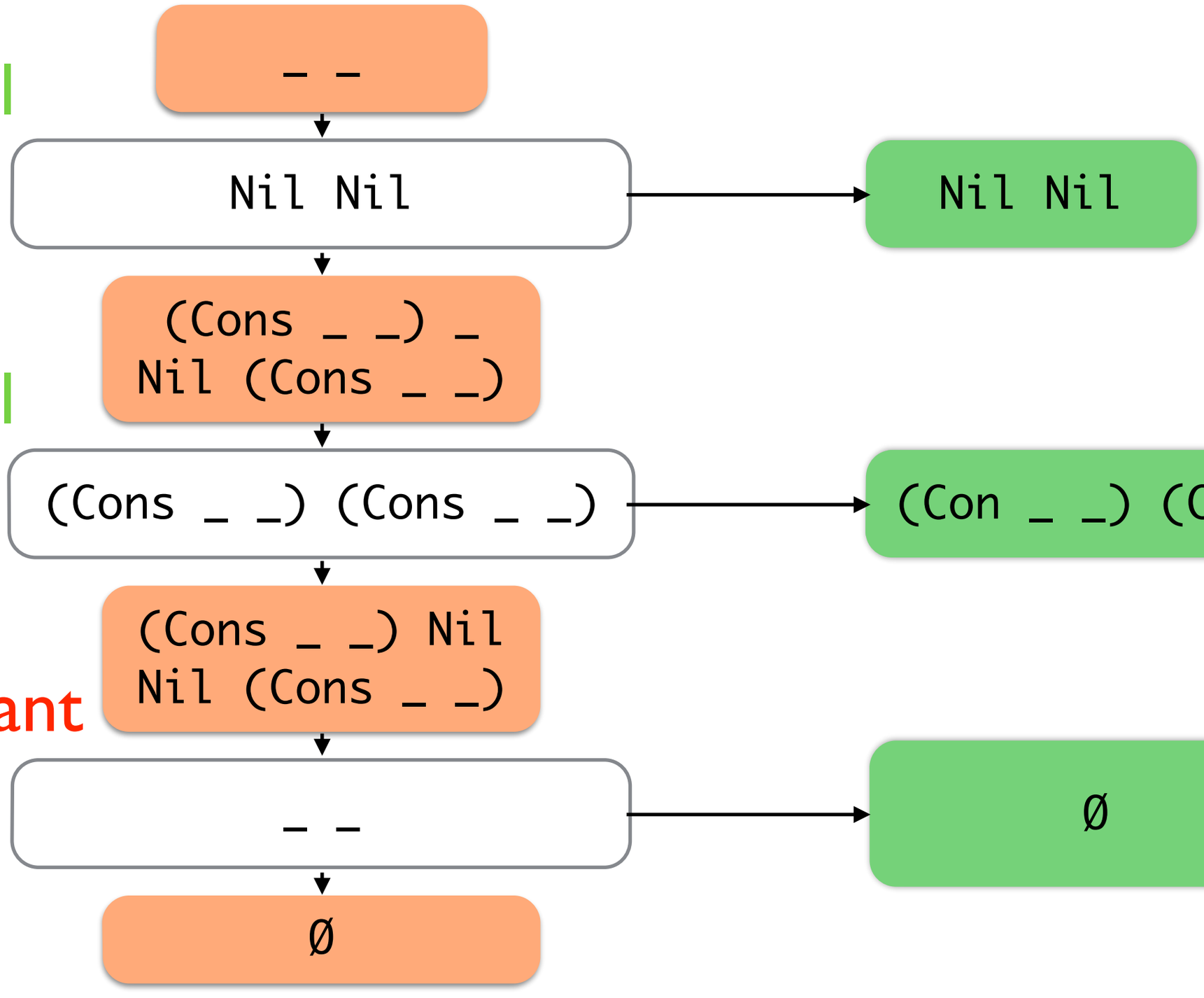
Type-checking

```
zip :: Vec n a -> Vec n b -> Vec n (a,b)
```




_ _
Nil Nil
Nil Nil
(Cons _ _) _
Nil (Cons _ _)
(Cons _ _) (Cons _ _)
(Con _ _) (Cons _ _)
(Cons _ _) Nil
Nil (Cons _ _)
_ _
Ø
Ø

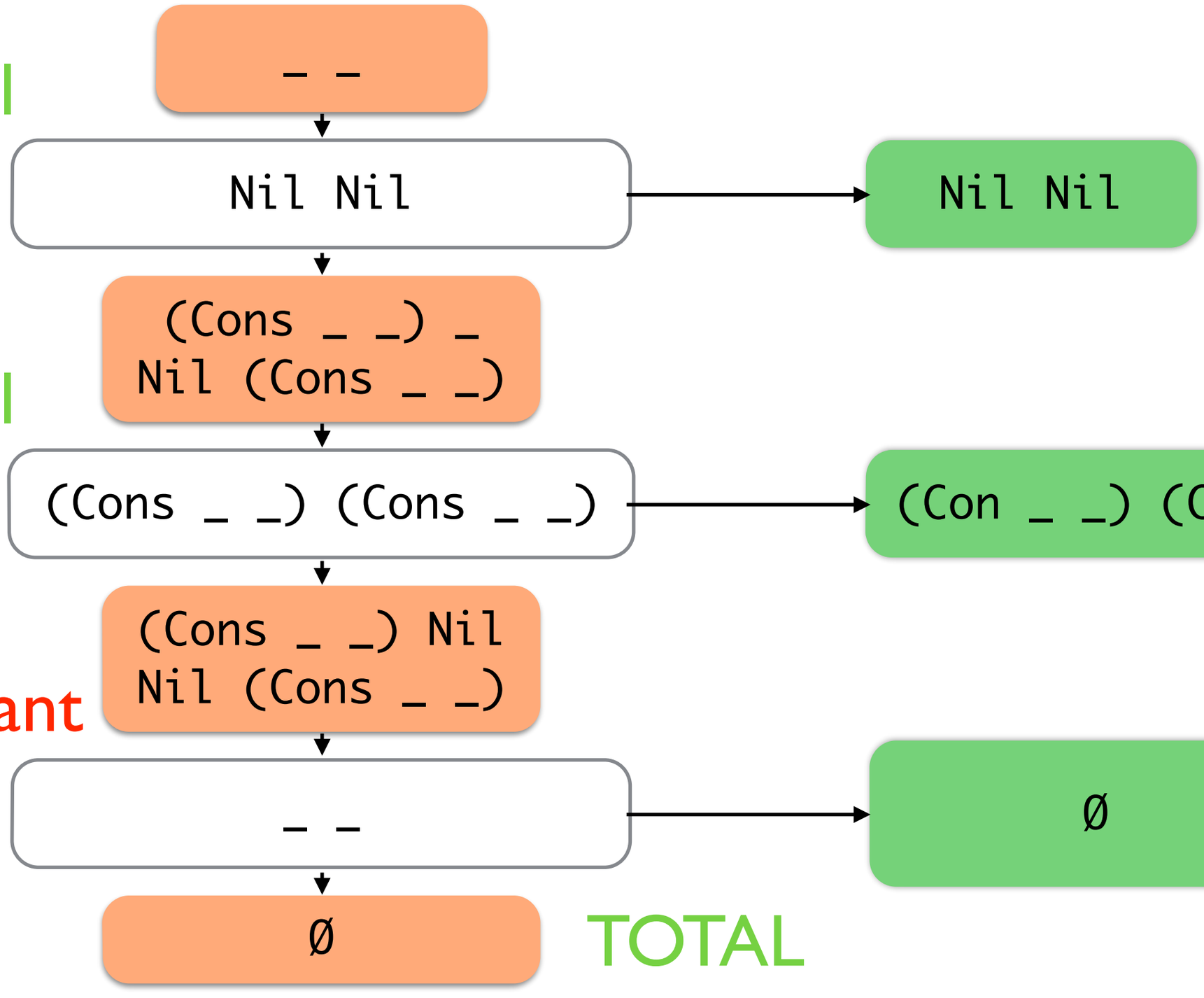
Useful



Useful

Redundant

Useful



Useful

Redundant

TOTAL

Summary

Discussed

- Pattern Matching in Haskell
- Basic static checks for pattern matching (totality & redundancy)
- Our incremental approach for detecting these problems
- Easy integration with type checking

As of Today

- Formalisation of the algorithm
- Prototype implementation in GHC
  - Support for guards & literals
  - Reasoning about laziness
  - Tested (GHC bootstrapping & testsuite)

Future Work

- Optimise implementation (space & time)
- Test it on Hackage libraries
- GHC Extensions
- Better support for guards
  - ESC Haskell (Dana N. Xu, 2006)
  - Catch (Neil Mitchell et al, 2008)
  - Liquid Types (Niki Vazou et al, 2014)

Questions?