Generic programming with ornaments and dependent types

Yorick Sijsling

supervisors
Wouter Swierstra
Johan Jeuring

The role of types in Haskell/Agda

Types in Haskell/Agda

- What does a ‘List A’ tell us?
  - It is a list
  - It has elements of type A
- A lot of static information

Types in Haskell/Agda

- Types are precise
- New types are easy to make
- “If it compiles it works”

  - Haskell programmers

Lists in Agda

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

- Infix notation of _::_

(In Haskell)

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

- Hidden arguments

```
take : ∀ { A } → (n : Nat) → List A → List A
take zero _            = []
take (suc n) []        = ?1
take (suc n) (x :: xs) = x :: take n xs
```

- Agda functions are total, no exceptions!

```
take : ∀ { A } → (n : Nat) → List A → List A
take zero _           = []
take (suc n) []       = ?1
take (suc n) (x :: xs) = x :: take n xs
```

- Agda functions are total, no exceptions!
- A default value?
  - Like an empty list
  - Bugs!
- Change the type to Maybe (List A) ?
  - Makes the call site responsible

```
take : ∀ {A} → (n : Nat) → List A → List A
take zero _          = []
take (suc n) []      = ?1
take (suc n) (x :: xs) = x :: take n xs
```

- Agda functions are total, no exceptions!
- A default value?
  - Like an empty ...
  - Bug...

More types!

- Change the type to Maybe (List A)?
  - Makes the call site responsible

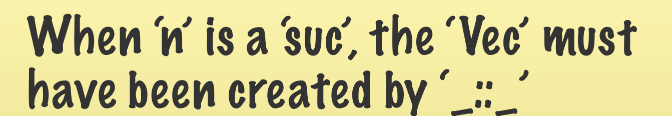
Inductive families

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

- Add a length index
- Different constructors produce different indices
- 'Vec A 0' is a different type than 'Vec A 3'

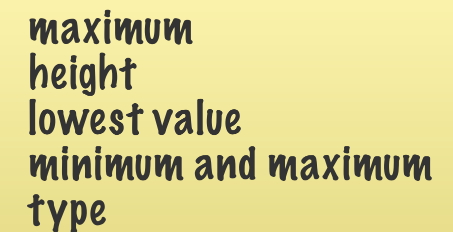
A family: for every length a family member

```
takeᵥ : ∀ {A m} → (n : Nat) →
  Vec A (n + m) → Vec A n
takeᵥ zero _ = []
takeᵥ (suc n) (x ∷ xs) = x ∷ takeᵥ n xs
```



- The failure can never occur
- Precise types == no bugs

More inductive fam



- Bounded trees (min/max indices)
- Balanced trees (height index)
- Ordered trees (lowest/highest indices)
- Red-black trees (color/black-depth indices)

Inductive families

- Properties of data can be encoded
- Program logic is unique for every program
- Specialised datatypes are *often not reusable*

Find for lists

```
find  :  List Nat →
   (P  :  Nat → Bool) → Maybe Nat
find [] P  =  nothing
find (x :: xs) P  =  if (P x) then (just x) else (find xs P)
```

Find for vectors

- Vectors are just lists with a length index

```
findv : ∀ { n } → Vec Nat n →
  (P : Nat → Bool) → Maybe Nat
findv [] P = nothing
findv (x :: xs) P = if (P x) then (just x) else (findv xs P)
```

- Copy-paste for other kinds of lists

```
find_b  :  ∀ { mx } → BoundedNatList mx →
   (P  :  Nat → Bool) → Maybe Nat
find_b [] P  =  nothing
find_b (x :: xs) P  =  if (P x) then (just x) else (find_b xs P)

find_s  :  ∀ { l } → SortedNatList l →
   (P  :  Nat → Bool) → Maybe Nat
find_s [] P  =  nothing
find_s (x :: xs) P  =  if (P x) then (ju
```

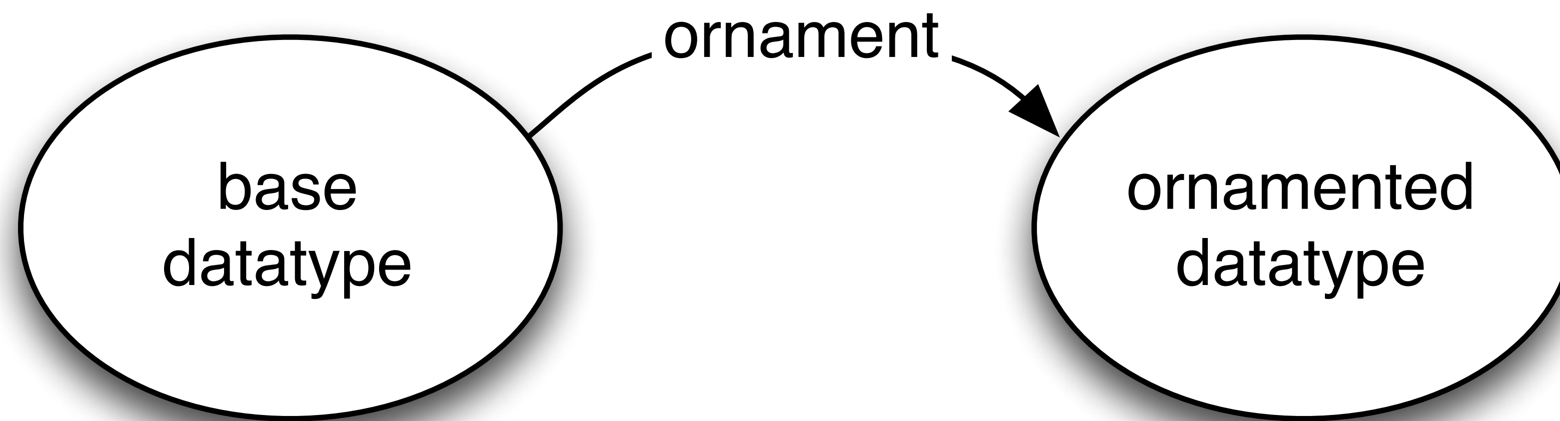
- Why did i include 4 definitions that are exactly the same?

- Inductive families have a huge design space
- Very specialised types are useful
- All these different types cause code duplication

Ornaments

Ornaments

- Express relations between datatypes

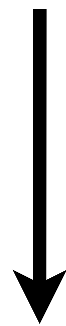


- Ornaments work as a patch on a base type

- They tell us how one datatype has to be modified to obtain another datatype.

Ornaments

```
data Nat  :  Set where
   zero  :  Nat
   suc   :  Nat → Nat
```



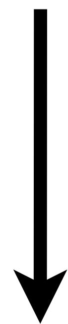
‘List is Nat with an element attached to every suc’

```
data List (A  :  Set)  :  Set where
   []   :  List A
   _::_   :  A → List A → List A
```

- Red boxes represent what the ornament needs to specify

Ornaments

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```



‘Vec is List with a length index’

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Ornaments

Characterisation of ornaments

- Elements of the ornamented type are at least as informative as elements of the base type
- Every ornament has a *forget* function
  - For nat→list? *length*
  - For list→vec? *vecToList*

Why ornaments?

- *Bring structure to the datatype design space*
- Potential uses:
  - Generating new datatypes
  - Implementation of functions
  - Refactoring
  - Interfacing with less strong type systems

Related to refinement types

Describing datatypes

Describing datatypes

- Datatype-generic programming
- *Descriptions* describe datatypes
- Assign a type to every description

Descriptions

```
data Desc : Set where
  '1  : Desc
  _⊕_ : Desc → Desc → Desc
  _⊗_ : Desc → Desc → Desc
```

Semantics

$$[\![_]\!]_{desc} : Desc \to Set$$
$$[\![\text{'1}]\!]_{desc} = \top$$
$$[\![A \oplus B]\!]_{desc} = Either\ [\![A]\!]_{desc}\ [\![B]\!]_{desc}$$
$$[\![A \otimes B]\!]_{desc} = [\![A]\!]_{desc} \times [\![B]\!]_{desc}$$

Sum and products
x is pair

Note: Agda prelude, not stdlib

Description of booleans

$$\begin{aligned}\mathsf{boolDesc} &: \mathsf{Desc} \\ \mathsf{boolDesc} &= `1 \oplus `1\end{aligned}$$

Semantics of boolDesc

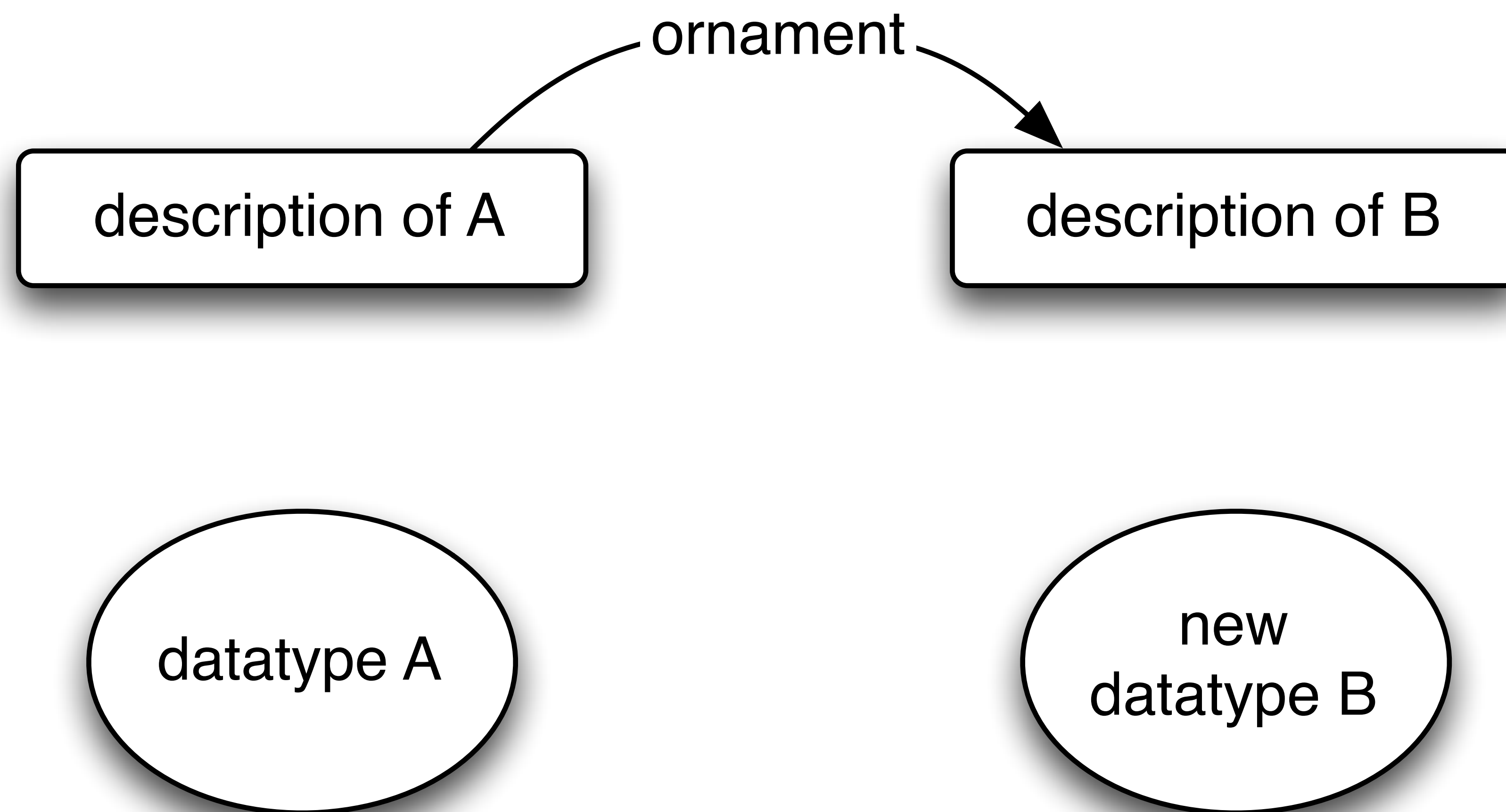
$$\begin{aligned}&[\![`1 \oplus `1]\!]_{desc} \\ &\quad \equiv \mathsf{Either}\ [\![`1 ]\!]_{desc}\ [\![ `1]\!]_{desc} \\ &\quad \equiv \mathsf{Either}\ \top\ \top\end{aligned}$$

Embedding-projection pair

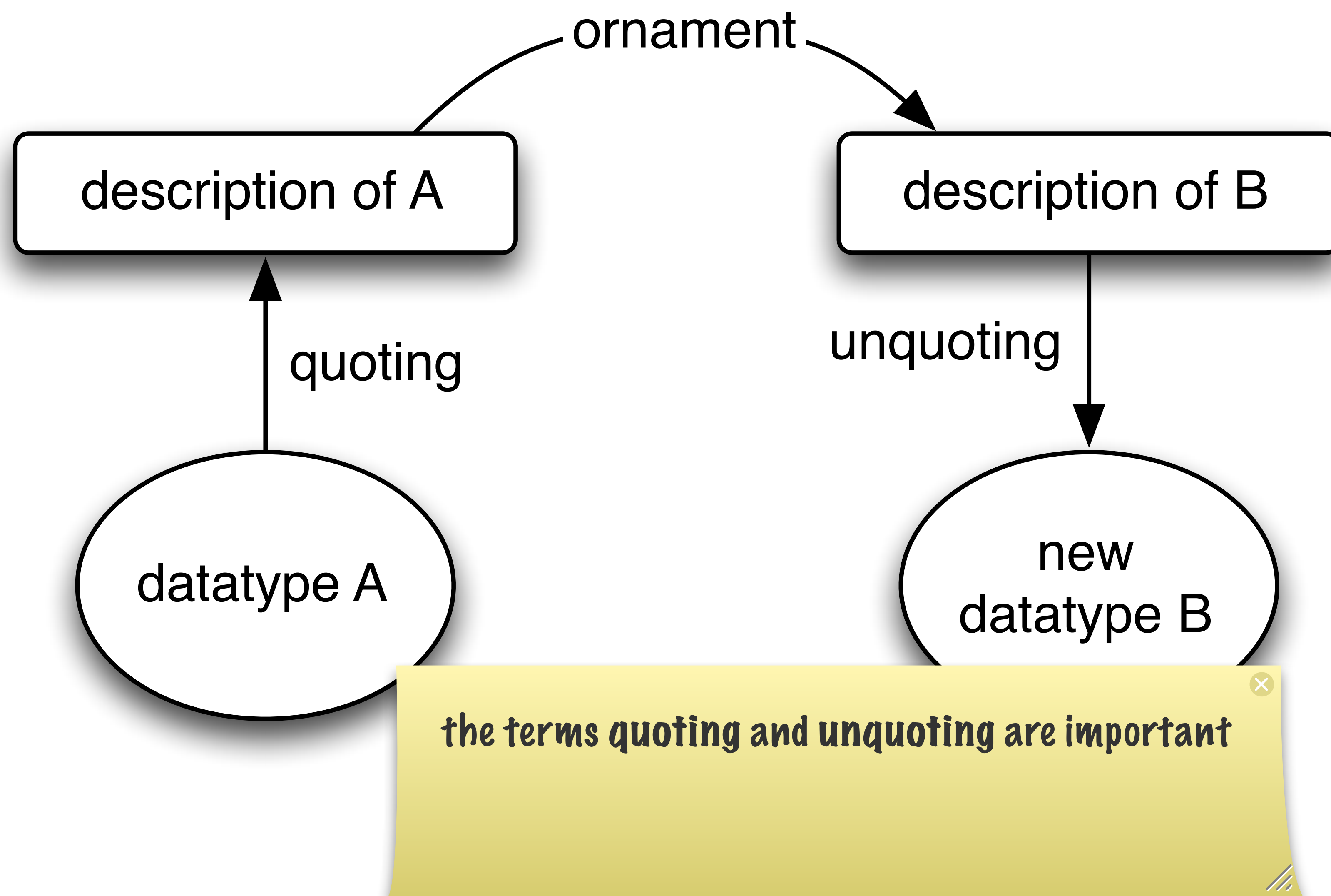
$$\text{bool-to} : \text{Bool} \to [\![\text{`1} \oplus \text{`1}]\!]_{desc}$$
$$\text{bool-to false} = \text{left tt}$$
$$\text{bool-to true} = \text{right tt}$$

$$\text{bool-from} : [\![\text{`1} \oplus \text{`1}]\!]_{desc} \to \text{Bool}$$
$$\text{bool-from (left tt)} = \text{false}$$
$$\text{bool-from (right tt)} = \text{true}$$

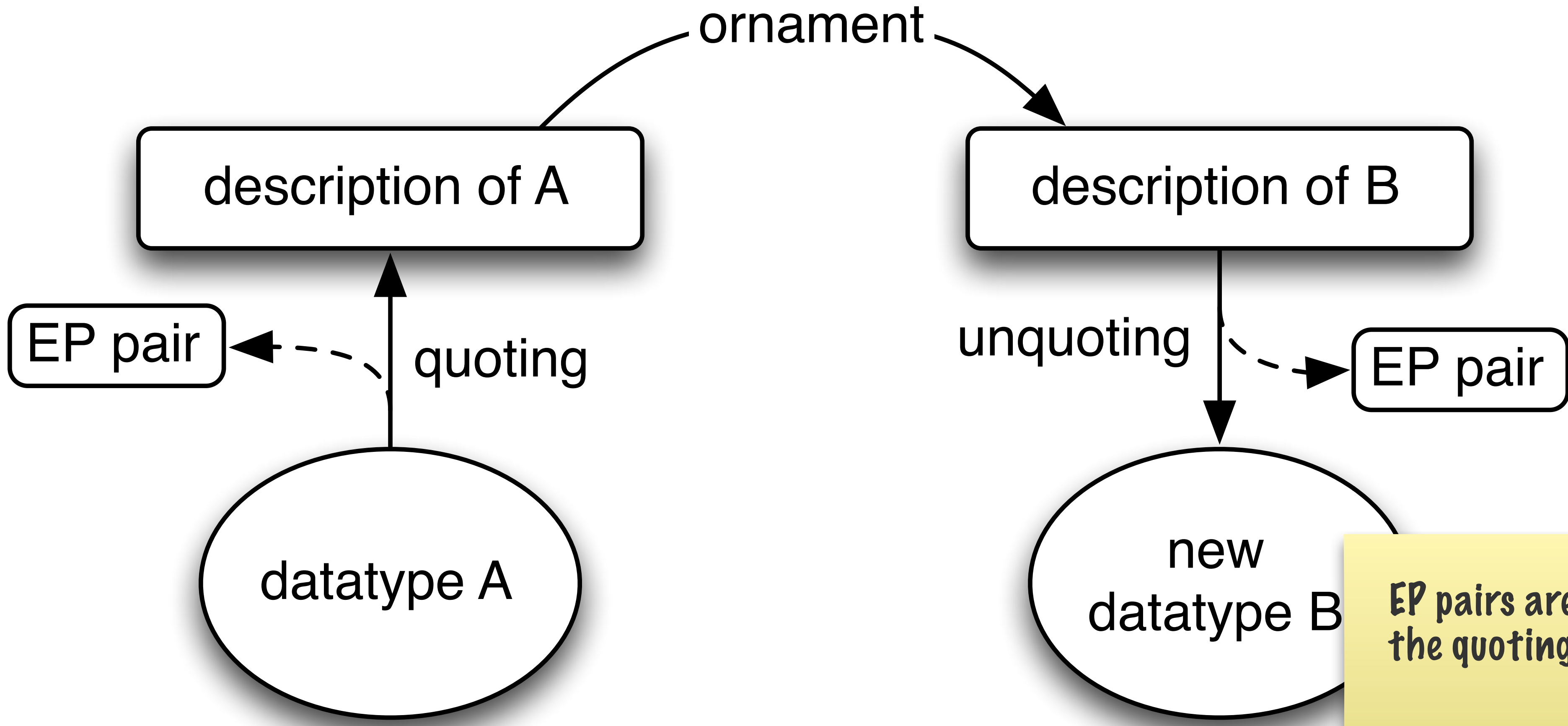
Ornaments work on descriptions



Reflection connects datatypes to descriptions

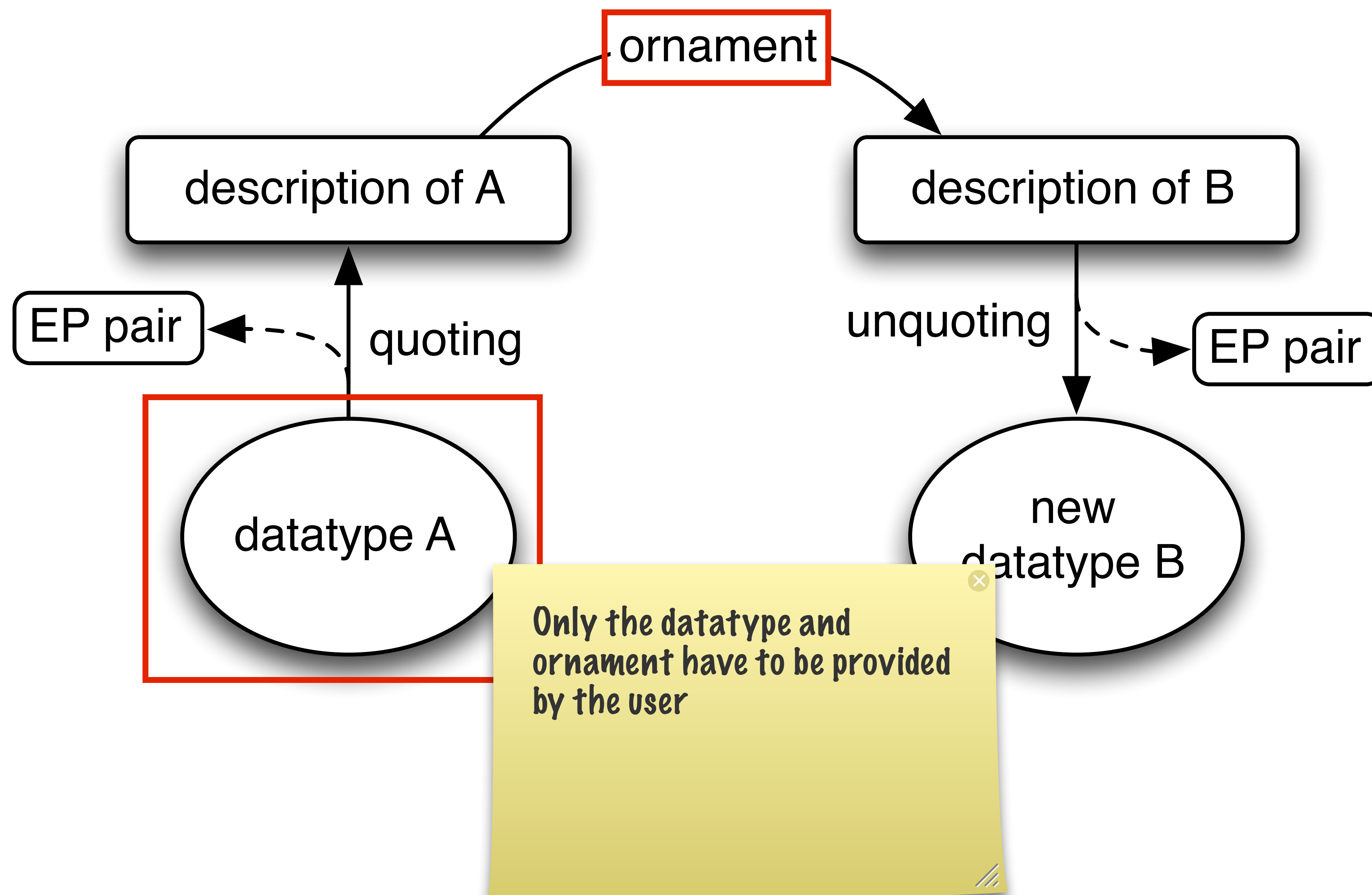


Reflection connects datatypes to descriptions



EP pairs are generated during the quoting/unquoting

Reflection connects datatypes to descriptions



- Descriptions need to be designed with unquoting in mind

- A pair of bools $(\text{`1} \oplus \text{`1}) \otimes (\text{`1} \oplus \text{`1})$ can not be written as one datatype!

Contributions

- A universe of descriptions featuring:
  - Parameters (as a telescope)
  - Indices (as a telescope)
  - Dependent types
- Always convertible to a datatype!

Contributions

- Ornaments for these descriptions
  - Copying, inserting, deleting of arguments
  - Refining of indices and parameters
- High-level ornaments like
  - ‘add a parameter argument’
  - ‘rename these arguments’

Contributions

- A generic programming framework
  - Quoting

  - Generic operations like fold and depth
  - Semi-automatic unquoting

The goal

- To provide a practical implementation of ornaments within Agda
- A framework for experimentation

24 minutes

PART 2
Results

We will show how the library works

Quoting

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

```
unquoteDecl quotedNat NatHasDesc =
  deriveHasDesc quotedNat NatHasDesc (quote Nat)
```

```
quotedNat  : QuotedDesc
NatHasDesc : HasDesc Nat
```

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

```
unquoteDecl quotedNat NatHasDesc =
  deriveHasDesc quotedNat NatHasDesc (quote Nat)
```

‘Quoting of Nat’

```
quotedNat : QuotedDesc
```

```
NatHasDesc : HasDesc Nat
```

```
quotedNat : QuotedDesc
```

- Contains some meta-information
- Contains a description

```
natDesc : Desc ε ε _
natDesc = QuotedDesc.desc quotedNat
```

quotedNat : QuotedDesc

- Contains some meta-information
- Contains a description

No indices

No parameters

```
natDesc  :  Desc ε ε _
natDesc  =  QuotedDesc.desc quotedNat
```

NatHasDesc : HasDesc Nat

- An *instance* of HasDesc
- Contains the embedding-projection pair
- Automatically found with instance search

```
natTo : Nat → μ natDesc tt tt
natTo = to

natFrom : μ natDesc tt tt → Nat
natFrom = from
```

to and from require an instance of HasDesc

'μ natDesc': elements for this description
'tt tt': instantiate parameters and indices (there are none)

```
data List (A : Set) : Set where
  nil  : List A
  cons : (x : A) → (xs : List A) → List A
```

‘Quoting of List’

```
quotedList : QuotedDesc
```

```
ListHasDesc : ∀ { A } → HasDesc (List A)
```

quotedList : QuotedDesc

One parameter of type Set

No indices

listDesc : Desc ε (ε ▷′ Set) _

listDesc = QuotedDesc.desc quotedList

```
ListHasDesc : ∀ { A } → HasDesc (List A)
```

- The HasDesc instance works for any A

```
listTo : ∀ { A } → List A → μ listDesc (tt , A) tt
listTo = to

listFrom : ∀ { A } → μ listDesc (tt , A) tt → List A
listFrom = from
```

```
ListHasDesc : ∀ {A} → HasDesc (List A)
```

gdepth is a generic function

```
gdepth : ∀ {A} → ⦃ R : HasDesc A ⦄ → A → Nat
```

It can calculate the length of lists

```
length : ∀ {A} → List A → Nat
length = gdepth
```

```
ListHasDesc : ∀ { A } → HasDesc (List A)
```

gdepth is a generic function

```
gdepth : ∀ { A } → ⦃ R : HasDesc A ⦄ → A → Nat
gdepth = gfold (depthAlg listDesc)
```

- Algebra is for a specific description
- depthAlg gives an algebra for any description

It can calculate the length of lists

```
length : ∀ { A } → List A → Nat
length = gdepth
```

Ornamentation

```
data Nat : Set where
  zero : Nat
  suc : (n : Nat) → Nat
```

Rename “n” to “xs”
Add parameter A
Add argument “x” of type A

```
data List (A : Set) : Set where
  nil : List A
  cons : (x : A) → (xs : List A) → List A
```

- Names of arguments are in the descriptions
- Other names are not

```
nat→list : Orn _ _ _ _ natDesc
nat→list = renameArguments 1 (just "xs" :: [])
  >>+ addParameterArg 1 "x"
```

```
nat→list  :  Orn _ _ _ _ natDesc
nat→list  =  renameArguments 1 (just ”xs” :: [])
   ≫⁺ addParameterArg 1 ”x”
```

It is an ornament on Nat

```
nat→list  :  Orn _ _ _ _ natDesc
nat→list  =  renameArguments 1 (just "xs" :: [])
   ≫⁺ addParameterArg 1 "x"
```

Compose two ornaments

```
nat→list  :  Orn _ _ _ _ natDesc
nat→list  =  renameArguments 1 (just ”xs” :: [])
  ≫⁺ addParameterArg 1 ”x”
```

Rename the argument of the second constructor to “xs”

```
nat→list  :  Orn _ _ _ _ natDesc
nat→list  =  renameArguments 1 (just "xs" :: [])
   >>+ addParameterArg 1 "x"
```

Add a type parameter to the datatype

Use it in the second constructor

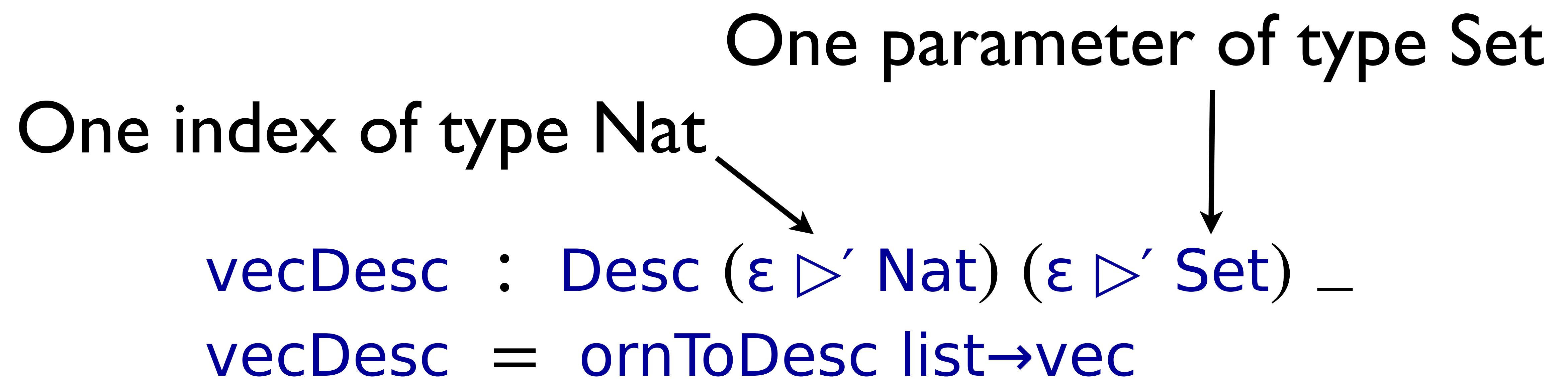
The ornament results in the description of lists

ornToDesc nat→list ≡ listDesc

Algebraic ornaments

- Length of lists is calculated with depthAlg
- Vectors are lists with a length index
- Use the algebra to calculate the index

```
list→vec  :  Orn _ _ _ _ listDesc
list→vec  =  algOrn (depthAlg listDesc)
```


One parameter of type Set
One index of type Nat
vecDesc : Desc (ε ▷′ Nat) (ε ▷′ Set) _
vecDesc = ornToDesc list→vec

Bonus

Descriptions

- Built with 5 components:

Unit type	ι I
Argument of type S	nm / S ⊗ xs
Inductive argument	nm /rec I ⊗ xs
Empty type	‘0
Constructor	x ⊕ xs

Description of vectors

```
vecDesc  :  DatDesc (ε ▷′ Nat) (ε ▷′ Set) _
vecDesc  =  ι (λ γ → (tt , 0))
    ⊕ ”n”  /  (λ γ → Nat)
    ⊗ ”x”  /  (λ γ → top (pop γ))
    ⊗ ”xs”  /rec (λ γ → tt , top (pop γ))
    ⊗ ι (λ γ → tt , suc (top (pop γ)))
    ⊕ ‘0
```

Nat index
Set parameter

```
vecDesc  :  DatDesc (ε ▷′ Nat) (ε ▷′ Set) _
vecDesc  =  ι (λ γ → (tt , 0))
   ⊕ ”n”  /  (λ γ → Nat)
   ⊗ ”x”  /  (λ γ → top (pop γ))
   ⊗ ”xs”  /rec (λ γ → tt , top (pop γ))
   ⊗ ι (λ γ → tt , suc (top (pop γ)))
   ⊕ ‘0
```

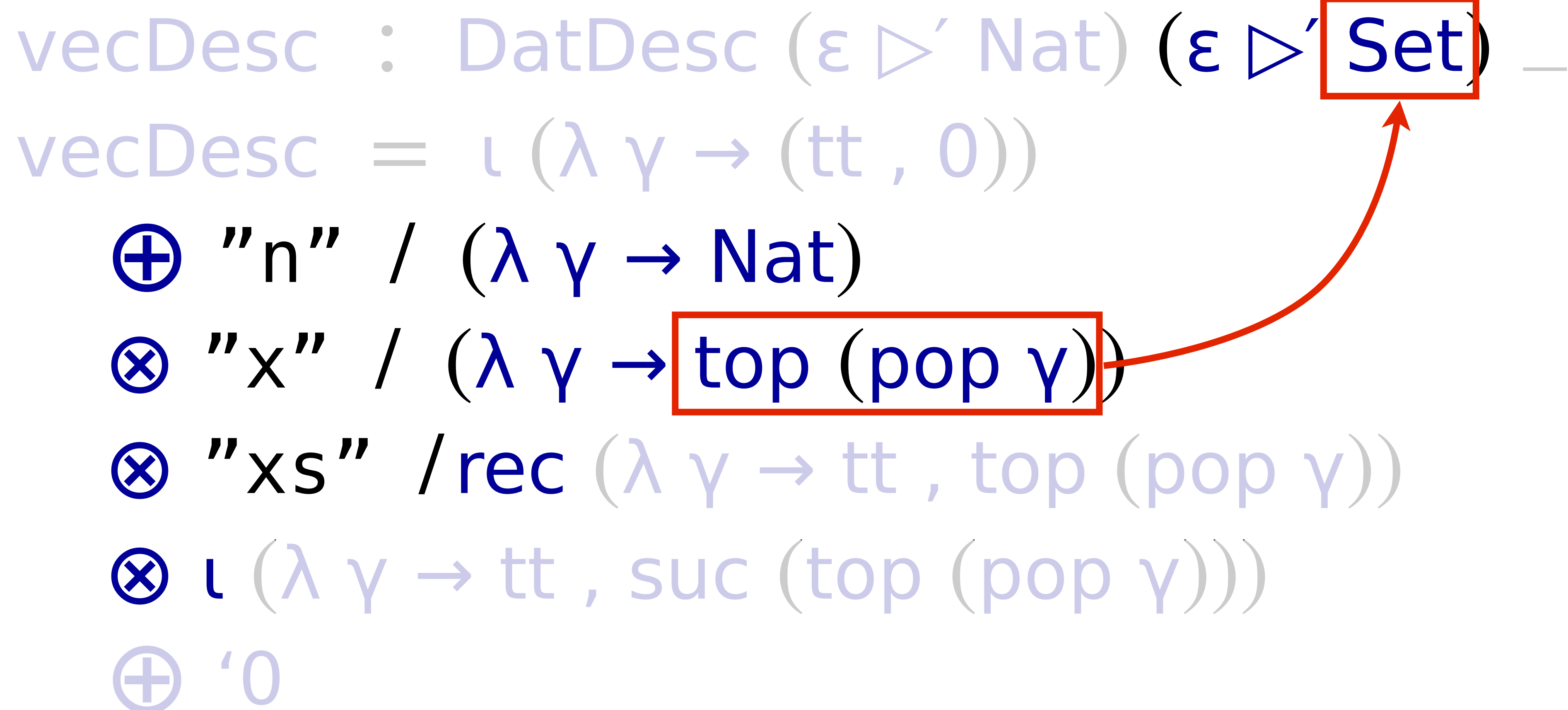
Nil constructor

```
vecDesc  :  DatDesc (ε ▷′ Nat) (ε ▷′ Set) _
vecDesc  =  ι (λ γ → (tt , 0))
   ⊕ ”n”  /  (λ γ → Nat)
   ⊗ ”x”  /  (λ γ → top (pop γ))
   ⊗ ”xs”  /rec (λ γ → tt , top (pop γ))
   ⊗ ι (λ γ → tt , suc (top (pop γ)))
   ⊕ ‘0
```

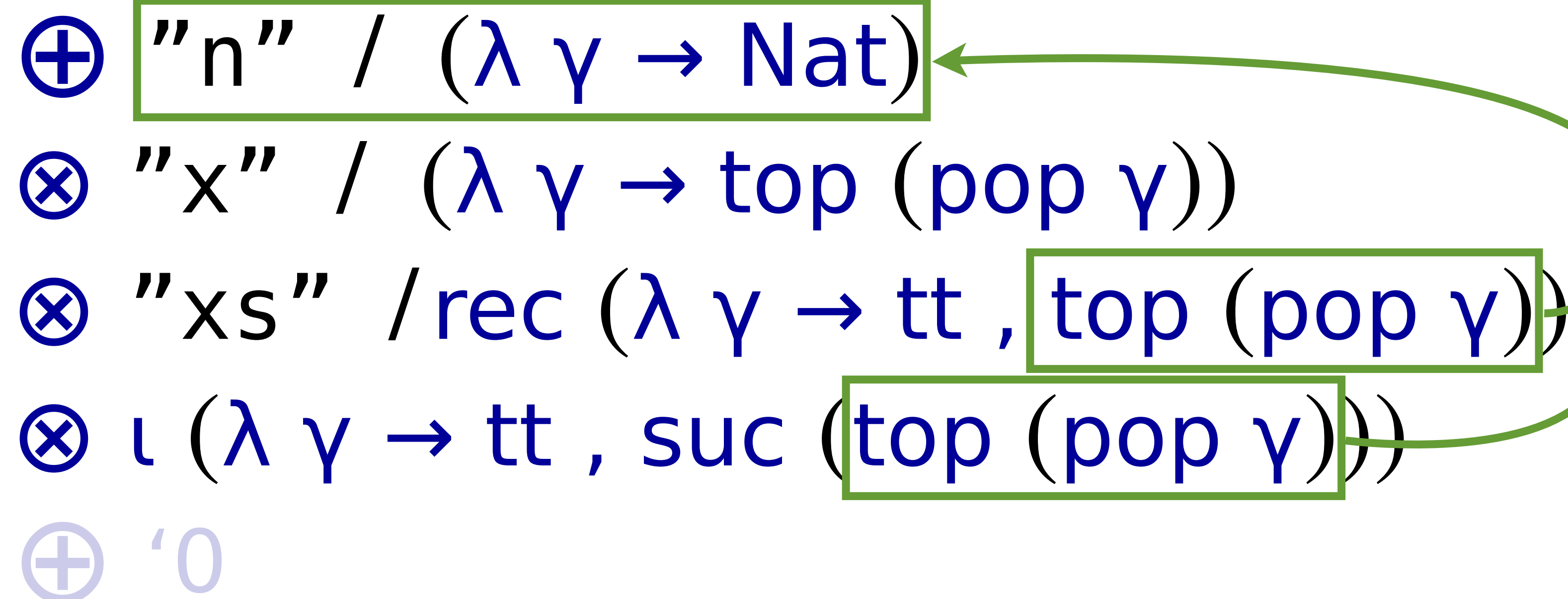
Cons constructor

```
vecDesc  :  DatDesc (ε ▷′ Nat) (ε ▷′ Set) _
vecDesc  =  ι (λ γ → (tt , 0))
   ⊕ "n"  / (λ γ → Nat)
   ⊗ "x"  / (λ γ → top (pop γ))
   ⊗ "xs" /rec (λ γ → tt , top (pop γ))
   ⊗ ι (λ γ → tt , suc (top (pop γ)))
   ⊕ ‘0
```

Cons constructor



Cons constructor



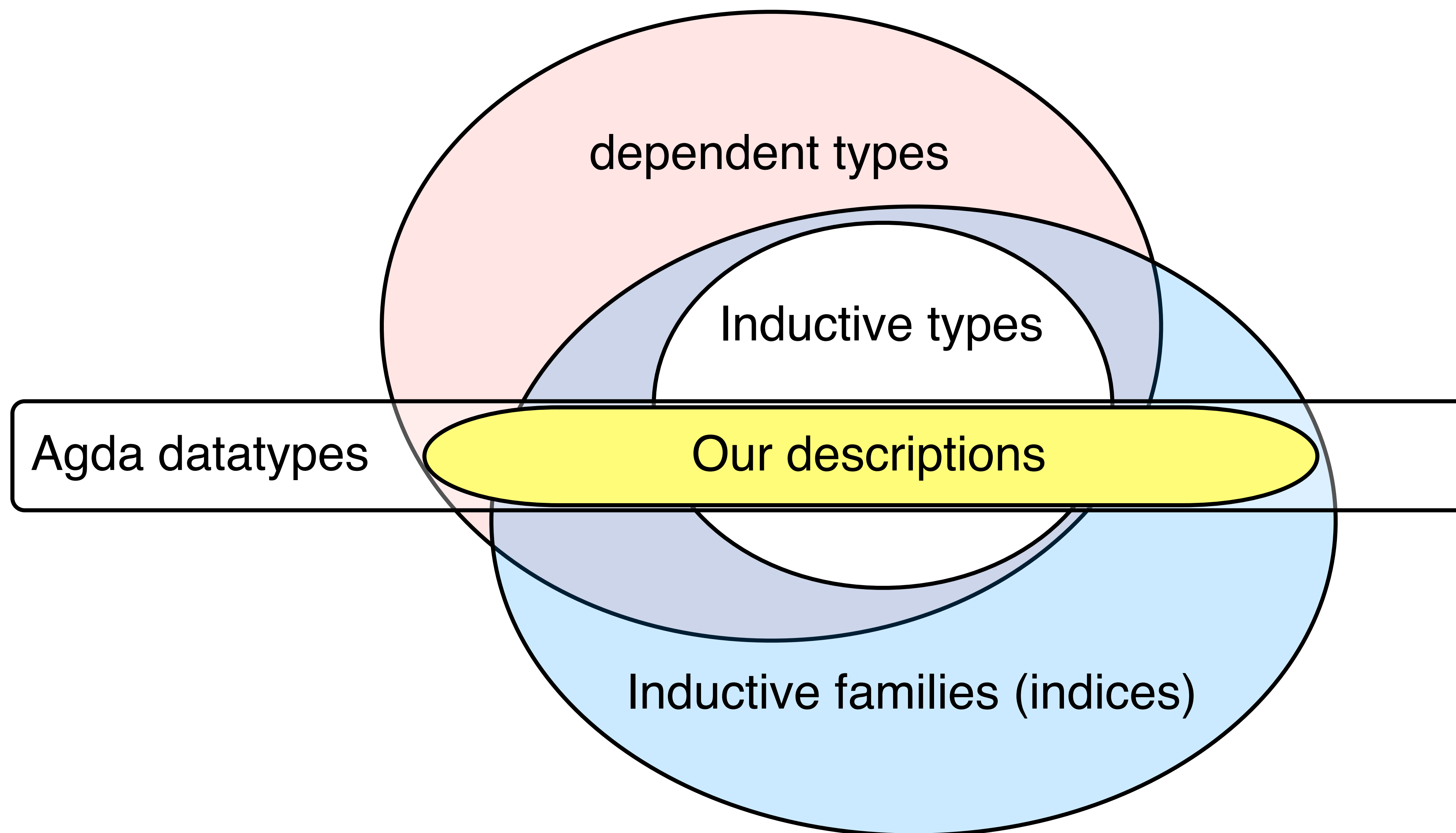
Terminate with empty type

```
vecDesc  :  DatDesc (ε ▷′ Nat) (ε ▷′ Set) _
vecDesc  =  ι (λ γ → (tt , 0))
   ⊕ ”n”  /  (λ γ → Nat)
   ⊗ ”x”  /  (λ γ → top (pop γ))
   ⊗ ”xs”  /rec (λ γ → tt , top (pop γ))
   ⊗ ι (λ γ → tt , suc (top (pop γ)))
   ⊕ ‘0
```

Conclusion

Conclusion

- Parameters, indices, dependent types
- Always convertible to an Agda datatype



Conclusion

Up until now, formalisations of ornaments could always produce types that could not be written as a datatype

- With the right descriptions and ornaments:

Agda datatypes are closed under ornamentation

- High-level ornaments can formalise intuitive ideas about relations between datatypes

'This description is that description with an extra parameter'
'This description is that description with an extra index'
Was already kind of shown, but not for a universe like this

Conclusion

A generic programming framework with quoting, ornamentation and unquoting can work well

Generic programming with ornaments and dependent types

Yorick Sijsling

Thesis, slides and source code
at sijsling.com