Modularising inductive families

Josh Ko & Jeremy Gibbons

Department of Computer Science
University of Oxford

Workshop on Generic Programming
18 September 2011, Tokyo, Japan

Internalism Externalism

Internalism

Constraints internalised in datatypes

```
data Vec (A : Set) : Nat → Set where
  []  : Vec A 0
  _::_ : (x : A) →
         {n : Nat} (xs : Vec A n) →
         Vec A (suc n)

x :: y :: z :: [] : Vec A 3
```

Internalist clarity

Constraints cleanly expressed and managed

```
zipWith3 :
  (f : A → B → C → D) →
  Vec A n → Vec B n → Vec C n → Vec D n
zipWith3 f []       []       []
  = []
zipWith3 f (x :: xs) (y :: ys) (z :: zs)
  = f x y z :: zipWith3 f xs ys zs
```

Internalist libraries

dreadful reusability/composability

```
insert : Nat → List Nat → List Nat
vinsert :
  Nat → Vec Nat n → Vec Nat (suc n)
```

```
sinsert :
  (x : Nat) → SList b → SList (b ⊓ x)

data SList : Nat → Set where
  nil  : ∀ {b} → SList b
  cons : (x : Nat) → ∀ {b} → b ≤ x →
         (xs : SList x) → SList b
```

Externalism

Predicates imposed on existing datatypes

```
(xs : List Nat) × Length n xs
```

```
data Length : Nat → List A → Set where
  nil  : Length 0 []
  cons : ∀ {x n xs} →
         Length n xs →
         Length (suc n) (x :: xs)
```

Externalist composability

Easy to impose multiple constraints

```
(xs : List Nat) × Length n xs × Sorted b xs
```

```
data Sorted : Nat → List Nat → Set where
  nil  : ∀ {b} → Sorted b []
  cons : ∀ {x b} → b ≤ x →
         ∀ {xs} → Sorted x xs →
         Sorted b (x :: xs)
```

Externalist composability

Ideal for structuring libraries

```
insert : Nat → List Nat → List Nat

insert-length :
  Length n xs →
  Length (suc n) (insert x xs)

insert-sorted :
  Sorted b xs →
  Sorted (b ⊓ x) (insert x xs)
```

Is it possible to import

externalist composability

into

internalist libraries ?

Or: Can we get sorted vectors and insert on sorted vectors for free?

	Constraints	Multiple constraints
Internalism	??	??
Externalism	Predicates	Pointwise conjunction of predicates

	Constraints	Multiple constraints
Internalism	Ornaments!	??
	Conor McBride	
Externalism	Predicates	Pointwise conjunction of predicates

Ornaments

Information added to a datatype to get a fancier one

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

Ornaments

Information added to a datatype to get a fancier one

```
data List (A : Set) : Nat → Set where
  []  : List A 0
  _::_ : (x : A) →
         {n : Nat} (xs : List A n) →
         List A (suc n)
```

Ornaments

Information added to a datatype to get a fancier one

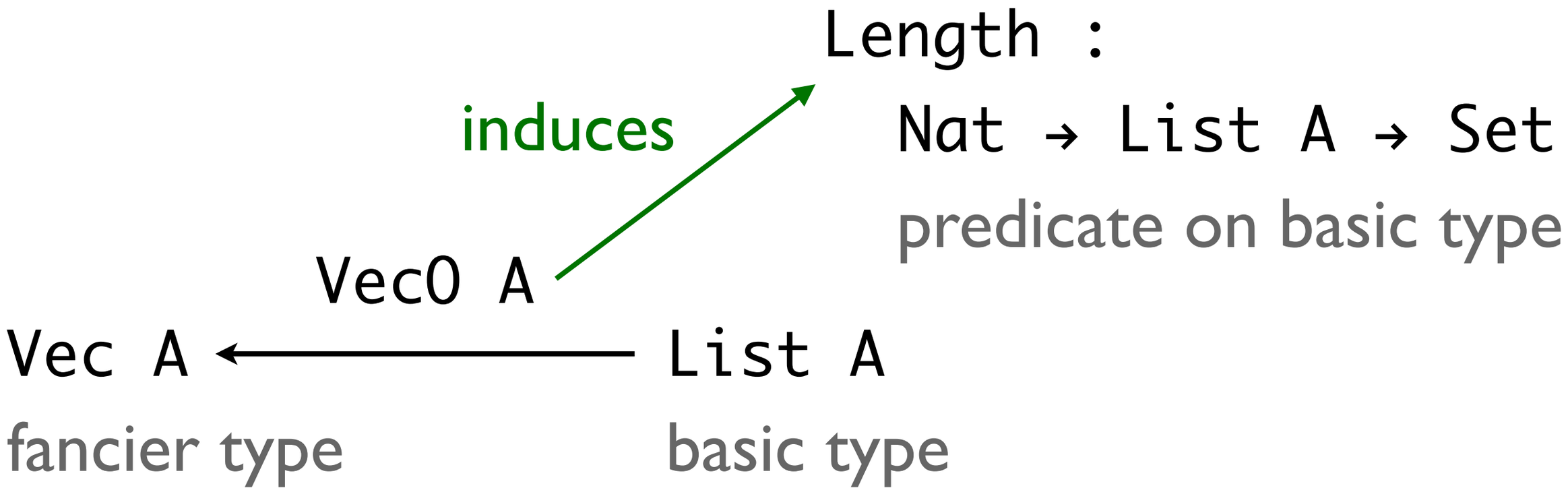
```
data Vec (A : Set) : Nat → Set where
  []  : Vec A 0
  _::_ : (x : A) →
         {n : Nat} (xs : Vec A n) →
         Vec A (suc n)
```

	Constraints	Multiple constraints
Internalism	Ornaments	??

induce

Externalism	Predicates	Pointwise conjunction of predicates

Ornaments induce predicates



Ornaments induce predicates

and corresponding isomorphisms

internalist

```
Vec A n  ≅  (xs : List A)   × Length n xs
```

fancier type

external

basic type

induced predicate

```
SList b  ≅  (xs : List Nat) × Sorted b xs
```

Function upgrade

with the help of the isomorphisms

```
Vec Nat n  ≅  (xs : List Nat) × Length n xs
```

```
vinsert : Nat    →
  Vec Nat n      → Vec Nat (suc n)
      ≀                  ≀
  xs : List Nat  ↦ insert x xs : List Nat
  l : Length n xs ↦ insert-length l :
                    Length (suc n) (insert x xs)
```

```
insert : Nat → List Nat → List Nat
insert-length :
  Length n xs → Length (suc n) (insert x xs)
```

Function upgrade

with the help of the isomorphisms

```
Vec Nat n  ≅  (xs : List Nat) × Length n xs
vinsert : Nat → Vec Nat n → Vec Nat (suc n)

SList b  ≅  (xs : List Nat) × Sorted b xs
sinsert : (x : Nat) → SList b → SList (b ⊓ x)
```

```
insert : Nat → List Nat → List Nat
insert-length :
  Length n xs → Length (suc n) (insert x xs)
insert-sorted :
  Sorted b xs → Sorted (b ⊓ x) (insert x xs)
```

Sorted vectors

```
data SList : Nat → Set where
  nil  : ∀ {b} → SList b
  cons : (x : Nat) → ∀ {b} → b ≤ x →
         SList x → SList b

data Vec Nat : Nat → Set where
  []  : Vec Nat 0
  _::_ : Nat →
         ∀ {n} → Vec Nat n → Vec Nat (suc n)
```

Sorted vectors

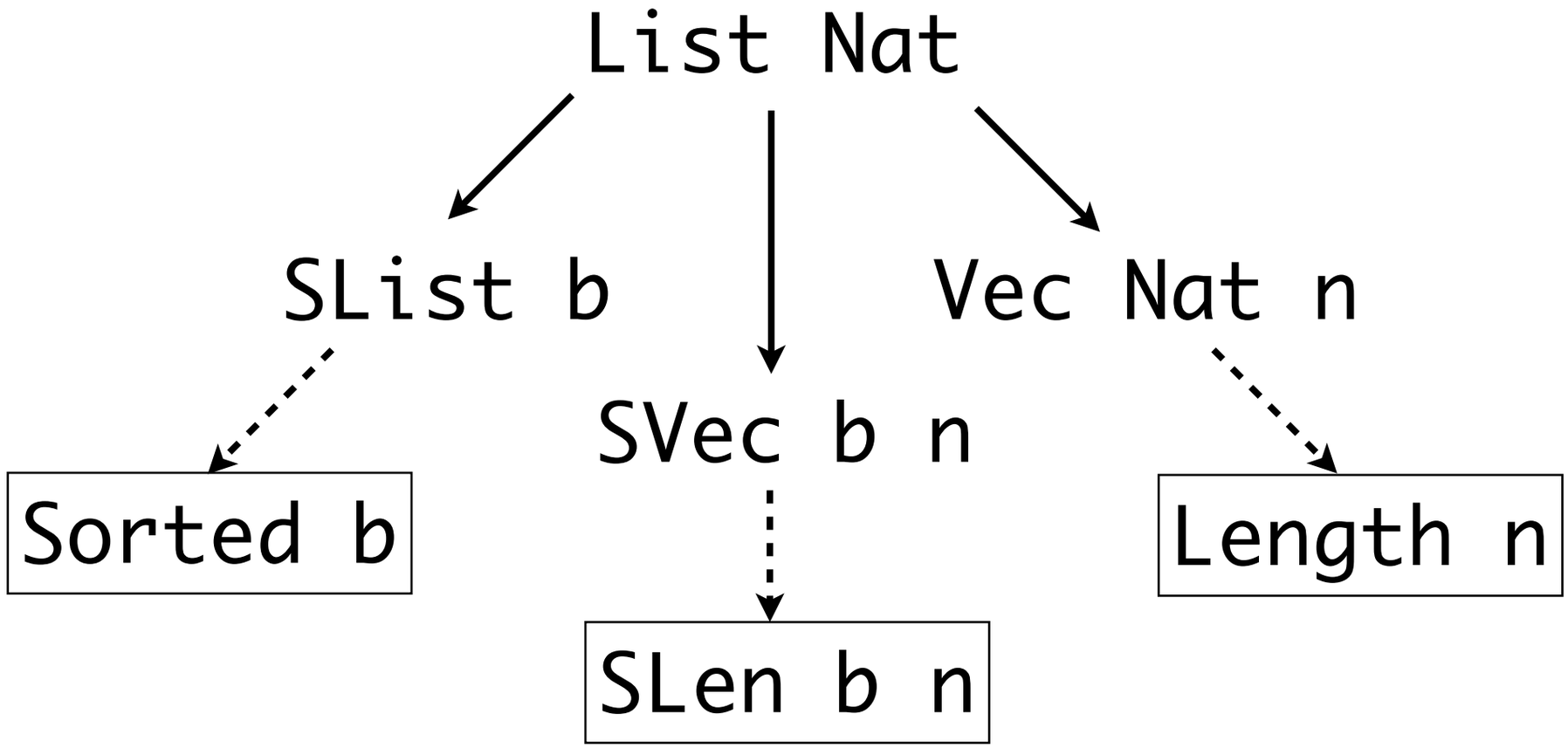
= sorted lists + vectors!

```
data SVec : Nat → Nat → Set where
  nil  : ∀ {b} → SVec b 0
  cons : (x : Nat) → ∀ {b} → b ≤ x →
         ∀ {n} → SVec x n → SVec b (suc n)
```

	Constraints	Multiple constraints
Internalism	Ornaments	Ornament fusion
	induce ↓	corresponds to ↓
Externalism	Predicates	Pointwise conjunction of predicates

Ornament fusion

corresponds to conjunction of induced predicates




```
SLen b n xs  ≅  Sorted b xs × Length n xs
```

Ornament fusion

corresponds to conjunction of induced predicates

```
    SVec b n
≅   (xs : List Nat) × SLen b n xs
≅   (xs : List Nat) × Sorted b xs
                    × Length n xs
```

Function upgrade

with the help of the isomorphisms

```
SVec b n  ≅  (xs : List Nat) × Sorted b xs
                             × Length n xs

svinsert : (x : Nat) →
  SVec b n          → SVec (b ⊓ x) (suc n)
     ≀                     ≀
  xs : List Nat     ↦ insert x xs : List Nat
  s : Sorted b xs   ↦ insert-sorted s :
                      Sorted (b ⊓ x) (insert x xs)
  l : Length n xs   ↦ insert-length l :
                      Length (suc n) (insert x xs)
```

Modular library structure

```
data List
..., insert, ...
```

```
orn Vec inducing Length
..., insert-length, ...
```

```
orn SList inducing Sorted
..., insert-sorted, ...
```

	Constraints	Multiple constraints
Internalism	Ornaments	Ornament fusion
	induce ↓	corresponds to ↓
Externalism	Predicates	Pointwise conjunction of predicates

Thanks!

Descriptions

A "universe" datatype containing codes for datatypes

```
Vec : Set → Desc Nat
Vec A =
  σ Bool (false↦ say zero
          true↦  σ A   λ x →
                 σ Nat λ n →
                 ask n * say (suc n))

μ : Desc I → (I → Set)
    -- μ (Vec A) : Nat → Set
```

Ornaments

A richer universe of *relative* descriptions

```
Vec0 : Set → Orn Nat ...

Vec0 A =

  σ Bool (false↦ say (ok zero)
          true↦  σ A    λ x →
                 Δ Nat λ n →
                 ask (ok n) * say (ok (suc n)))

⌞_⌟ : Orn J ... → Desc J

  -- ⌞ Vec0 A ⌟ ≈ Vec A
```