Accepted for WGP'11

Modularising inductive families

Josh Ko & Jeremy Gibbons

Department of Computer Science
University of Oxford

Dependently Typed Programming workshop
27 August 2011, Nijmegen, The Netherlands

Internalism

Constraints internalised in datatypes

```
data Fin : Nat → Set where
  zero : ∀ {m} → Fin (suc m)
  suc  : ∀ {m} → Fin m → Fin (suc m)
```

Externalism

Predicates imposed on existing datatypes

```
(n : Nat) × (n < m)
  -- Σ Nat (λ n ↦ n < m)

data _<_ : Nat → Nat → Set where
  base : ∀ {m} → zero < suc m
  step : ∀ {m n} → n < m → suc n < suc m
```

Internalism vs. Externalism

An isomorphism. Coincidence?

```
Fin m  ≅  (n : Nat) × (n < m)
```

Internalism vs. Externalism

An isomorphism — no coincidence!

```
Fin m  ≅  (n : Nat) × (n < m)
```

```
data Fin : Nat → Set where
  zero : ∀ {m} → Fin (suc m)
  suc  : ∀ {m} → Fin m → Fin (suc m)
```

```
data _<_ : Nat → Nat → Set where
  base : ∀ {m} → zero < suc m
  step : ∀ {m n} → n < m → suc n < suc m
```

Internalism vs. Externalism

An isomorphism — no coincidence!

```
Fin m  ≅  (n : Nat) × (n < m)
```

```
data Fin : Nat → Set where
  zero : ∀ {m} → Fin (suc m)
  suc  : ∀ {m} → Fin m → Fin (suc m)
```

```
data _<_ : Nat → Nat → Set where
  base : ∀ {m} → zero < suc m
  step : ∀ {m n} → n < m → suc n < suc m
```

Conor McBride's *ornamentation*

Internalism vs. Externalism

An isomorphism — no coincidence!

```
Fin m  ≅  (n : Nat) × (n < m)

data Fin : Nat → Set where
  zero : ∀ {m} → Fin (suc m)
  suc  : ∀ {m} → Fin m → Fin (suc m)

data _<_ : Nat → Nat → Set where
  base : ∀ {m} → zero < suc m
  step : ∀ {m n} → n < m → suc n < suc m
```

Conor McBride's *algebraic ornamentation*

Algebraic ornamentation

To index the type of `xs` with `foldr f e xs` ...

```
f : A → B → B
e : B
```

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

Algebraic ornamentation

To index the type of `xs` with `foldr f e xs` ...

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

Algebraic ornamentation

To index the type of `xs` with `foldr f e xs` …

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

```
foldr f e [] ≡ e
```

Algebraic ornamentation

To index the type of `xs` with `foldr f e xs` ...

```
data List : B → Set where
  []   : List e
  _∷_  : (x : A) →
         {b : B} (xs : List b) →
         List
```

```
foldr f e [] ≡ e
```

Algebraic ornamentation

To index the type of `xs` with `foldr f e xs` ...

```
data List : B → Set where
  []   : List e
  _∷_  : (x : A) →
         {b : B} (xs : List b) →
         List (f x b)
```

```
foldr f e [] ≡ e
```

```
foldr f e (x ∷ xs) ≡ f x (foldr f e xs)
                   ≡ f x b
```

Algebraic ornamentation

To index the type of `xs` with `length xs` ...

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

```
length [] ≡ zero
```

```
length (x ∷ xs) ≡ suc (length xs)
                ≡ suc n
```

```
List A  ≅  (n : Nat) × Vec A n
```

Internalism vs. Externalism

An isomorphism — no coincidence!

```
Fin m  ≅  (n : Nat) × (n < m)
```

```
data Fin : Nat → Set where
  zero : ∀ {m} → Fin (suc m)
  suc  : ∀ {m} → Fin m → Fin (suc m)
```

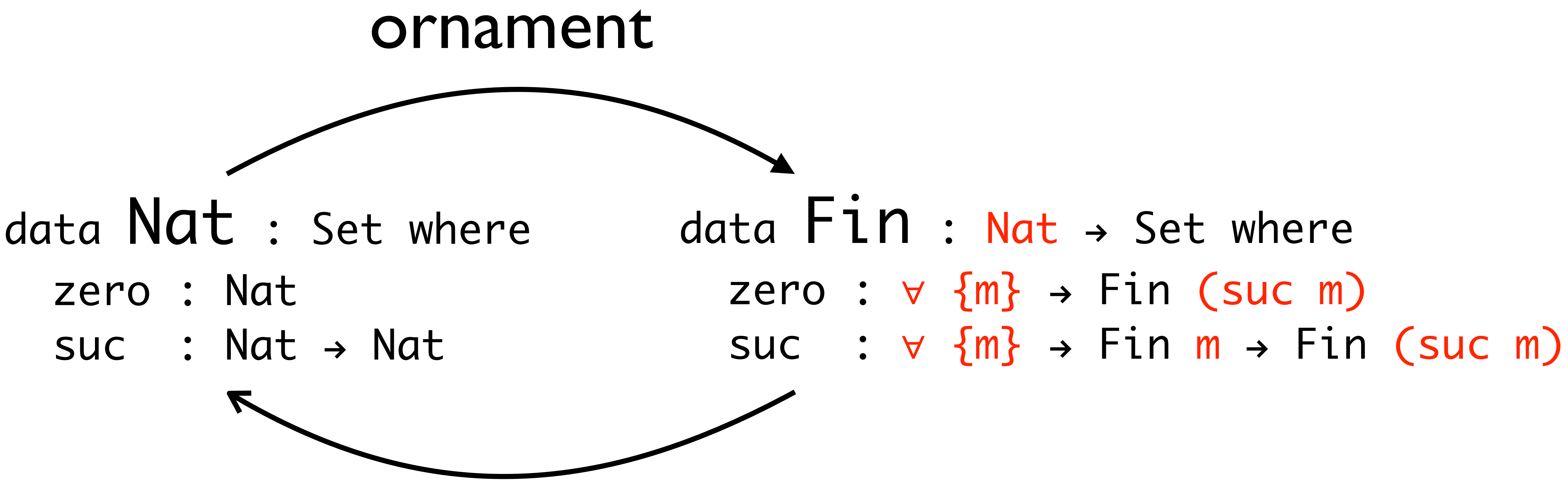
```
data _<_ : Nat → Nat → Set where
  base : ∀ {m} → zero < suc m
  step : ∀ {m n} → n < m → suc n < suc m
```

ornamental-

Conor McBride's *algebraic ornamentation*

Datatype-generically

An ornament induces a predicate & an isomorphism.



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

```
data Fin : Nat → Set where
  zero : ∀ {m} → Fin (suc m)
  suc  : ∀ {m} → Fin m → Fin (suc m)
```

```
forget : Fin m → Nat
forget zero    = zero
forget (suc i) = suc (forget i)
```

Datatype-generically

An ornament induces a predicate & an isomorphism.

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

```
data Fin : Nat → Set where
  zero : ∀ {m} → Fin (suc m)
  suc  : ∀ {m} → Fin m → Fin (suc m)
```

underlying natural number

```
data _<_ : Nat → Nat → Set where
  base : ∀ {m} → zero < suc m
  step : ∀ {m n} → n < m → suc n < suc m
```

```
Fin m  ≅  (n : Nat) × (n < m)
```

Example: vectors

vectors = lists with length information

```
data List (A : Set) : Set
```

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

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

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

Example: sorted lists

sorted lists indexed with a lower bound

```
data List Nat : Set
[]  : List Nat
_::_ : Nat →
      List Nat → List Nat
```

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

Example: sorted lists

sorted lists indexed with a lower bound

```
data List Nat : Set
[]   : List Nat
_::_ : Nat →
       List Nat → List Nat
```

```
data SList : Nat → Set where
  []   : ∀ {b} → SList b
  _::_ : (x : Nat) → ∀ {b} → b ≤ x →
         SList x → SList b
```

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

```
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 : ∀ {x n xs} →
  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 : ∀ {x n xs} →
  Length n xs → Length (suc n) (insert x xs)
insert-sorted : ∀ {x b xs} →
  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 zero
  _::_ : Nat →
           ∀ {n} → Vec Nat n → Vec Nat (suc n)
```

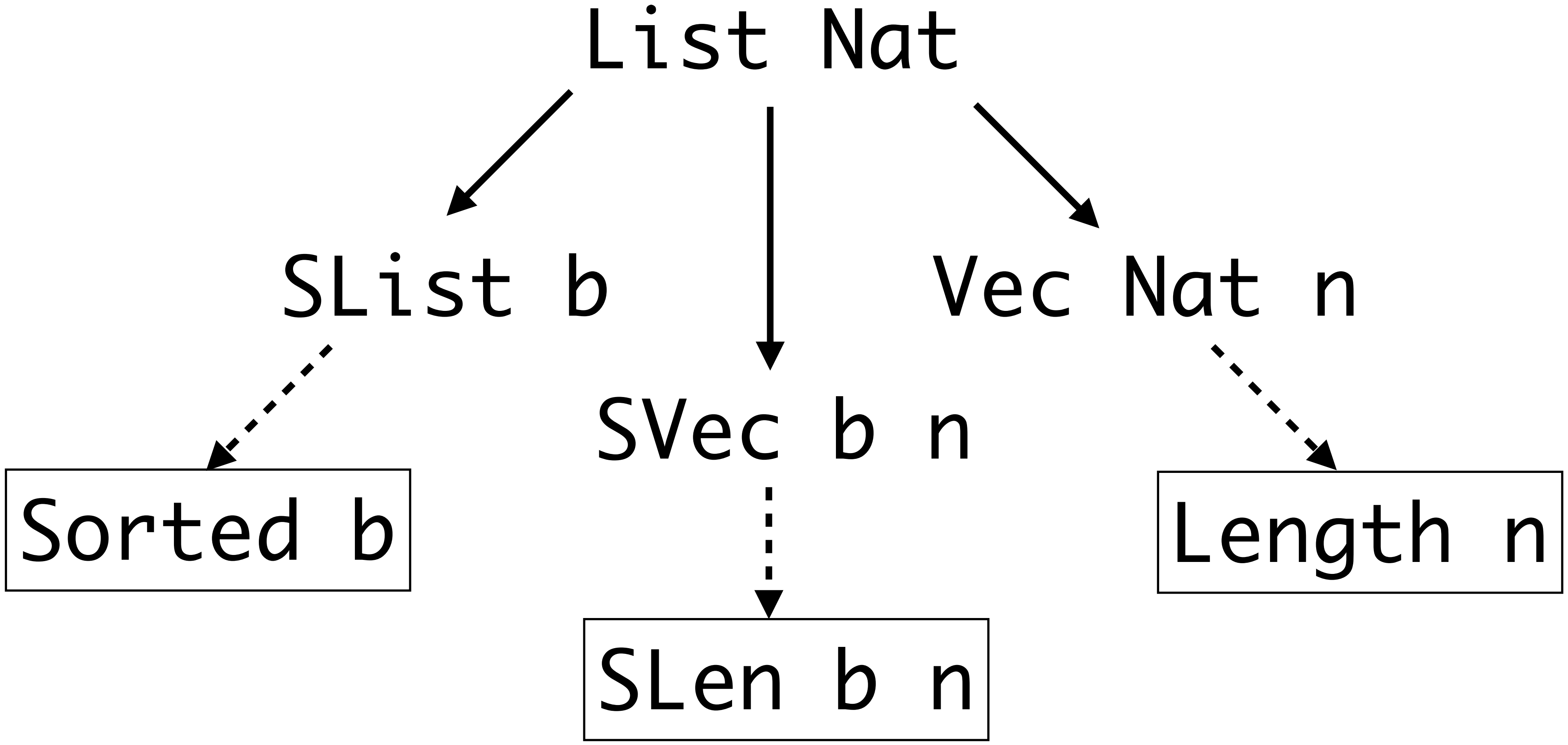
Sorted vectors

= sorted lists + vectors!

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

Ornament fusion

corresponds to conjunction of induced predicates


List Nat

SList b

SVec b n

Vec Nat n

Sorted b

SLen b n

Length n

`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)
```

Summary

It’s all about exploiting the connection between internalism and externalism.

Summary

- Datatype-generically, an ornament induces a predicate and an isomorphism — a raw object satisfying the predicate can be converted to a richer object via the isomorphism.
- Functions whose properties are proved externally can be upgraded to an internalist version with the help of the isomorphisms.

Summary

- Ornaments can be fused to integrate multiple constraints into a single datatype; fusion of ornaments corresponds to pointwise conjunction of induced predicates.
- To upgrade a function to work with a type synthesised out of composite ornamentation, relevant properties can be proved separately (and reused later).

Thanks!

Please read our WGP paper!

Another perspective...

Function upgrade — really worth the effort?

```
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 : ∀ {x n xs} →
  Length n xs → Length (suc n) (insert x xs)
```

Composability

Had we followed the more direct path...

```
svinsert : (x : Nat) →
  SVec b n         → SVec (b ⊓ x) (suc n)
      ↧                      ??
  xs : SList b     ↦ sinsert x xs : SList (b ⊓ x)
  ys : Vec Nat n   ↦ vinsert x ys : Vec Nat (suc n)
```

The integration doesn't go through —
unless the underlying lists can be shown to be the same.

```
sinsert : (x : Nat) → SList b → SList (b ⊓ x)
vinsert : Nat → Vec Nat n → Vec Nat (suc n)
```

Pre-/post-conditions

Index bounded by list length

```
lookup : ∀ {A} → (xs : List A)
               → (i : Nat) → i < length xs
               → A

lookup : ∀ {A} → ∀ {n}
               → (xs : Vec A n)
               → (i : Fin n)
               → A
```

Pre-/post-conditions

Same underlying data

```
integrate :
  (xs : SList b) (ys : Vec Nat n) →
  forget xs ≡ forget ys → SVec b n

integrate : ∀ {xs} →
  Sorted b xs → Length n xs → SLen b n xs
```

Need to expose underlying data as index — ornamental-algebraic ornamentation does exactly this (and does it systematically).