Generic Numerical Representations as Ornaments

From Number System to Datastructure

Samuel Klumpers

Utrecht University

What does that mean

The functional programmer has many datastructures at their disposal, to name some:

- linked lists
- random-access lists
- (skew) binomial heaps
- finger trees
- ...

What does that mean

The functional programmer has many datastructures at their disposal, to name some:

linked lists		unary naturals
random-access lists		binary
(skew) binomial heaps	“look like”	(skew) binary
finger trees		two—sided binary
…		

…and a bunch strongly resemble number systems.

What does that mean

The functional programmer has many datastructures at their disposal, to name some:

linked lists		unary naturals
random-access lists		binary
(skew) binomial heaps	“look like”	(skew) binary
finger trees		two—sided binary
…		

…and a bunch strongly resemble number systems.

“*This analogy can be exploited to design new implementations of container abstractions … Call an implementation designed in this fashion a **numerical representation**.*” (Okasaki, 1998)

What does that mean

The functional programmer has many datastructures at their disposal, to name some:

linked lists		unary naturals
random-access lists		binary
(skew) binomial heaps	"look like"	(skew) binary
finger trees		two—sided binary
...		

...and a bunch strongly resemble number systems.

"*This analogy can be exploited to design new implementations of container abstractions ... Call an implementation designed in this fashion a **numerical representation**.*" (Okasaki, 1998)

This raises the question: do "all" number systems have numerical representations?

Generic Numerical Representations

To answer the question:

- We specify the number systems and datastructures of interest with a *universe*;
- We express “X resembles Y”—statements using *ornaments*;
- Using the universe, we implement *generic programs*, sending number systems to their numerical representations.

Generic Numerical Representations

To answer the question:

- We specify the number systems and datastructures of interest with a *universe*;
- We express “X resembles Y”—statements using *ornaments*;
- Using the universe, we implement *generic programs*, sending number systems to their numerical representations.

Generic programming $\approx$ Programs that write programs

$\Rightarrow$ Write a number system, get a datastructure (and some of its operations/properties) for free.

We write and formalize these constructions in the language Agda.

Example: Lists

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

A list is either **empty**, or has **one more element** than another list.

Example: Lists

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

A list is either **empty**, or has **one more element** than another list.

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

A natural number is either **zero**, or is **one more** than another number.

Example: Lists look like naturals

How long is a list?

Example: Lists look like naturals

How long is a list?

```
length : List A → Nat
length []       = zero
length (x :: xs) = suc (length xs)
```

Example: Lists look like naturals

How long is a list?

```
length : List A → Nat
length []       = zero
length (x ∷ xs) = suc (length xs)
```

⇒ Lists simply become numbers by dropping their fields.

Example: Lists look like naturals

How long is a list?

```
length : List A → Nat
length []        = zero
length (x :: xs) = suc (length xs)
```

⇒ Lists simply become numbers by dropping their fields.

We can use length to relate functions between `Nat` and `List`.

If we have addition and concatenation

```
_+_  : Nat → Nat → Nat
_++_ : List A → List A → List A
```

then we expect that:

```
length (xs ++ ys) = length xs + length ys
```

Numerical Representations

In the other direction we can design the datastructure *after* the number system:

number system N	$\longrightarrow$	numerical representation T
successor	$\Rightarrow$	prepend
addition	$\Rightarrow$	concatenation
subtraction	$\Rightarrow$	lookup

Numerical Representations

In the other direction we can design the datastructure *after* the number system:

number system N	$\longrightarrow$	numerical representation T
successor	$\Longrightarrow$	prepend
addition	$\Longrightarrow$	concatenation
subtraction	$\Longrightarrow$	lookup

How do we make this precise?

Numerical Representations

In the other direction we can design the datastructure *after* the number system:

number system N	$\longrightarrow$	numerical representation T
successor	$\Longrightarrow$	prepend
addition	$\Longrightarrow$	concatenation
subtraction	$\Longrightarrow$	lookup

How do we make this precise?

```
forget : T A → N
```

At the very least, we need a function **forget** to act like **length** for T A and N

Numerical Representations

In the other direction we can design the datastructure *after* the number system:

number system N	$\rightarrow$	numerical representation T
successor	$\Rightarrow$	prepend
addition	$\Rightarrow$	concatenation
subtraction	$\Rightarrow$	lookup

How do we make this precise?

```
forget : T A → N
```

At the very least, we need a function **forget** to act like **length** for T A and N

…which also should "preserve" some structure $\Rightarrow$ ornaments.

Before Ornaments: Universes

A *universe* is a type whose values represent other (data)types.

Before Ornaments: Universes

A *universe* is a type whose values represent other (data)types.

Universes: statements about datatypes $\longrightarrow$ statements about values.

Before Ornaments: Universes

A *universe* is a type whose values represent other (data)types.

Universes: statements about datatypes $\longrightarrow$ statements about values.

We can describe a universe by giving two parts:

- a datatype of codes, `U : Type`,
- a decoding, `⟦_⟧ : U → Type`.

Before Ornaments: Universes

A *universe* is a type whose values represent other (data)types.

Universes: statements about datatypes $\longrightarrow$ statements about values.

We can describe a universe by giving two parts:

- a datatype of codes, `U : Type`,
- a decoding, `⟦_⟧ : U → Type`.

Generic programming and proving inside U becomes ordinary induction on U.

How do we use universes?

```
data Con–rec : Type where
  𝟙 :        Con–rec                 -- end
  σ : Type → Con–rec → Con–rec  -- field
  ρ :        Con–rec → Con–rec  -- recursive field

U–rec = List Con–rec
```

= simple recursive datatypes. (Datatypes are just lists of constructors anyway...)

How do we use universes?

```
data Con—rec : Type where
  𝟙 :          Con—rec                -- end
  σ : Type → Con—rec → Con—rec   -- field
  ρ :          Con—rec → Con—rec   -- recursive field

U—rec = List Con—rec
```

= simple recursive datatypes. (Datatypes are just lists of constructors anyway...)

For example, we can interpret:

Con—rec	$\longrightarrow$	Constructor
`𝟙`	$\Rightarrow$	`zero : Nat`, or `[] : List A`
`ρ 𝟙`	$\Rightarrow$	`suc : Nat → Nat`
`σ A (ρ 𝟙)`	$\Rightarrow$	`_::_ : A → List A → List A`

How do we use universes?

```
NatD : U-rec
NatD = 𝟙         -- zero : Nat
     ∷ ρ 𝟙       -- suc  : Nat → Nat
     ∷ []

ListD : Type → U-rec
ListD A = 𝟙              -- []   : List A
        ∷ σ A (ρ 𝟙)      -- _∷_  : A → List A → List A
        ∷ []
```

How do we use universes?

```
NatD : U–rec
NatD = 𝟙       -- zero : Nat
     ∷ ρ 𝟙     -- suc  : Nat → Nat
     ∷ []

ListD : Type → U–rec
ListD A = 𝟙             -- []   : List A
        ∷ σ A (ρ 𝟙)     -- _∷_  : A → List A → List A
        ∷ []
```

(The only difference between `NatD` and `ListD` is the added field `σ A`!)

Ornaments

```
data Orn : U-rec → Type
```

We use ornaments (McBride, 2011) to describe when one datatype can be seen as another datatype with *extra structure*.

$\Rightarrow$ We can say that `Orn D` is the "type of patches" on top of `D`.

Ornaments

```
data Orn : U–rec → Type
```

We use ornaments (McBride, 2011) to describe when one datatype can be seen as another datatype with *extra structure*.

⇒ We can say that `Orn D` is the “type of patches” on top of `D`.

We expect to be able to apply a patch

```
toDesc    : Orn D → U
```

but also to be able to revert a patch

```
ornForget : (OD : Orn D)
          → ⟦ toDesc OD ⟧ → ⟦ D ⟧
```

How do we use ornaments?

```
data Orn : U-rec → Type where
  []  :                          Orn []
  _∷_ : ConOrn CD → Orn D → Orn (CD ∷ D)
```

(Ornaments are lists of constructor ornaments.)

How do we use ornaments?

```
data Orn : U-rec → Type where
  []   :                        Orn []
  _::_ : ConOrn CD → Orn D → Orn (CD :: D)
```

(Ornaments are lists of constructor ornaments.)

```
data ConOrn : Con-rec → Type where
  𝟙 :                               ConOrn 𝟙
  σ : (A : Type) → ConOrn CD → ConOrn (σ A CD)
  ρ :              ConOrn CD → ConOrn (ρ   CD)
  Δ : Type       → ConOrn CD → ConOrn CD
```

The patches 𝟙, σ and ρ represent “no changes”, while Δ patches a new field into a datatype.

How do we use ornaments?

This lets us express that lists have the structure of naturals:

Ornament	:	NatD	$\longrightarrow$	ListD A
𝟙	:	𝟙	$\Longrightarrow$	𝟙
Δ A (ρ 𝟙)	:	ρ 𝟙	$\Longrightarrow$	σ A (ρ 𝟙)

```
ListOD : Type → Orn NatD
ListOD A = 𝟙          -- : List A
         ∷ Δ A (ρ 𝟙) -- : A → List A → List A
         ∷ []
```

Binary numbers

Observations:

- lists can be constructed *from* naturals as an ornament
- `ornForget` relates lists and naturals
- $\Rightarrow$ lists are the numerical representation of naturals

Can we generalize these?

Binary numbers

Observations:

- lists can be constructed *from* naturals as an ornament
- `ornForget` relates lists and naturals
- $\Rightarrow$ lists are the numerical representation of naturals

Can we generalize these?

Let's take a look at (zeroless/bijective) binary numbers:

```
data Bin : Type where
  0b    :       Bin
  1b 2b : Bin → Bin
```

Binary numbers

We can evaluate these to natural numbers:

```
value : Bin → Nat
value 0b     = 0
value (1b n) = 2 * (value n) + 1
value (2b n) = 2 * (value n) + 2
```

E.g., we can represent 9 as "121", or `1b (2b (1b 0b))`, because

$$
\begin{aligned}
& \text{value}\,(1\text{b}\,(2\text{b}\,(1\text{b}\,0\text{b}))) \\
&= 2 \cdot (2 \cdot (2 \cdot 0 + 1) + 2) + 1 \\
&= 2^2 \cdot 1 + 2^1 \cdot 2 + 2^0 \cdot 1 \\
&= 9
\end{aligned}
$$

random-access lists look like binary numbers

`Random`, the numerical representation of `Bin`, also has three constructors:

```
data Random (A : Type) : Type where
  Zero :                                Random A
  One  : A     → Random (A × A) → Random A
  Two  : A → A → Random (A × A) → Random A
```

random-access lists look like binary numbers

`Random`, the numerical representation of `Bin`, also has three constructors:

```
data Random (A : Type) : Type where
  Zero :                                Random A
  One  : A     → Random (A × A) → Random A
  Two  : A → A → Random (A × A) → Random A
```

- `Zero` is just empty,
- `One x xs` holds double the elements of `xs`, plus one element `x`,
- `Two` is analogous, but plus two elements.

random-access lists look like binary numbers

Random, the numerical representation of Bin, also has three constructors:

```
data Random (A : Type) : Type where
  Zero :                                  Random A
  One  : A     → Random (A × A) → Random A
  Two  : A → A → Random (A × A) → Random A
```

- Zero is just empty,
- One x xs holds double the elements of xs, plus one element x,
- Two is analogous, but plus two elements.

```
size (One x xs) = 1b (size xs)
```

random-access lists look like binary numbers

`Random`, the numerical representation of `Bin`, also has three constructors:

```
data Random (A : Type) : Type where
  Zero :                               Random A
  One  : A     → Random (A × A) → Random A
  Two  : A → A → Random (A × A) → Random A
```

- `Zero` is just empty,
- `One x xs` holds double the elements of `xs`, plus one element `x`,
- `Two` is analogous, but plus two elements.

```
size (One x xs) = 1b (size xs)
```

`Random A` contains a field of `Random (A × A)`

⇒ `Random` is a nested type

We need a new Universe 1

However, `U–rec` can't describe parameters, let alone nested types...

We need a new Universe 1

However, `U–rec` can't describe parameters, let alone nested types...

To make matters worse, consider n—ary numbers:

```
data _–ary (n : Nat) : Type where
  0b :                    n –ary
  nb : Fin n → n –ary → n –ary
```

We need a new Universe 1

However, `U-rec` can't describe parameters, let alone nested types...

To make matters worse, consider n—ary numbers:

```
data _-ary (n : Nat) : Type where
  0b :                   n -ary
  nb : Fin n → n -ary → n -ary
```

The numerical representation has to have the same structure

```
data NRandom (n : Nat) (A : Type) : Type where
  Zero :                               NRandom A
  Some : (k : Fin n) → ? → ... → NRandom A
```

but the variable k "suddenly" changes the type we have to write at the question mark.

We need a new Universe 1

However, `U–rec` can't describe parameters, let alone nested types...

To make matters worse, consider n—ary numbers:

```
data _–ary (n : Nat) : Type where
  0b :                   n –ary
  nb : Fin n → n –ary → n –ary
```

The numerical representation has to have the same structure

```
data NRandom (n : Nat) (A : Type) : Type where
  Zero :                             NRandom A
  Some : (k : Fin n) → ? → ... → NRandom A
```

but the variable k "suddenly" changes the type we have to write at the question mark.

$\Rightarrow$ Two problems, one solution: *extensible telescopes* (Cockx and Escot, 2022)

A Universe using Telescopes

Telescopes are (dependent) lists of types, which are either empty $\varnothing$, or consist of another telescope extended by some type $\_\triangleright\_$.

A Universe using Telescopes

Telescopes are (dependent) lists of types, which are either empty `∅`, or consist of another telescope extended by some type `_▷_`.

Context	—	`Γ`
Instantiation	—	`⟦ Γ ⟧tel`
Value in context	—	`⟦ Γ ⟧tel → S`

A Universe using Telescopes

Telescopes are (dependent) lists of types, which are either empty `∅`, or consist of another telescope extended by some type `_▷_`.

Context	—	`Γ`
Instantiation	—	`⟦ Γ ⟧tel`
Value in context	—	`⟦ Γ ⟧tel → S`

With telescopes, we can define a typical universe which supports:

- parameters
- variables
- indices

Example: Telescopes

For example

Type		Telescope
`List (A : Type)`	—	`∅ ▷ const Type`
`NRandom (n : Nat) (A : Type)`	—	`∅ ▷ const Nat ▷ const Type`

Example: Telescopes

For example

Type	—	Telescope
`List (A : Type)`	—	`∅ ▷ const Type`
`NRandom (n : Nat) (A : Type)`	—	`∅ ▷ const Nat ▷ const Type`

As a simple example, we can describe lists as

```
ListD : Desc (∅ ▷ const Type)
ListD = 𝟙              -- List A
      ∷ σ par (ρ 𝟙) -- A → List A → List A
```

where `par` abbreviates the extractor for the last parameter of a telescope.

Example: Telescopes

For example

Type		Telescope
`List (A : Type)`	—	`∅ ▷ const Type`
`NRandom (n : Nat) (A : Type)`	—	`∅ ▷ const Nat ▷ const Type`

As a simple example, we can describe lists as

```
ListD : Desc (∅ ▷ const Type)
ListD = 𝟙               -- List A
      ∷ σ par (ρ 𝟙) -- A → List A → List A
```

where `par` abbreviates the extractor for the last parameter of a telescope.

By encoding variables as extensions of parameters, `Some` can access `(n : Nat)`. The field `(k : Fin n)` in `Some` then extends `∅` to `∅ ▷ λ (n , _) → (Fin n)`.

Extending the Universe

Our paper extends this to

```
data Desc (Me : Meta) (Γ : Tel) : Type
```

by adding

- nested types
- metadata
- composite types ⇒ we also get fingertrees
- flexible variable binding ⇒ non-dependent fields

(and dealing with the fallout in the folds and ornaments.)

Nested types

Now that a field σ can pull types out of a telescope Γ, we can let a recursive field ρ modify the telescope using a map of telescopes $\Gamma \to \Gamma$.

Nested types

Now that a field σ can pull types out of a telescope Γ, we can let a recursive field ρ modify the telescope using a map of telescopes $\Gamma \to \Gamma$.

For example, if $\Gamma = \varnothing \triangleright$ `const Type`, then $\Gamma \to \Gamma$ just becomes `Type → Type`.

Nested types

Now that a field σ can pull types out of a telescope Γ, we can let a recursive field ρ modify the telescope using a map of telescopes Γ → Γ.

For example, if Γ = ∅ ▷ const Type, then Γ → Γ just becomes Type → Type.

We can then describe Random by changing A to Double A = A × A

```
RandomD : Desc _ (∅ ▷ const Type)
RandomD = 𝟙                    -- Zero : Random A
        ∷ σ par                -- One  : A
        ( ρ (Double ∘ par) -- → Random (A × A)
          𝟙)                   -- → Random A
        ∷ σ (Double ∘ par) -- Two  : A × A
        ( ρ (Double ∘ par) -- → Random (A x A)
          𝟙)                   -- → Random A
```

Metadata: Number Systems

Our generic construction needs to know how to interpret a number system; `value : Bin → Nat` isn't enough:

- A function can't open up another function definition.
- (We don't have the time to try infinitely many values.)

Metadata: Number Systems

Our generic construction needs to know how to interpret a number system; `value : Bin → Nat` isn't enough:

- A function can't open up another function definition.
- (We don't have the time to try infinitely many values.)

A simple scheme of annotations, `Number`, describes a generalization of (dense) positional number systems:

`𝟙 {n}`		constantly n
`σ S {f}`	acts as	given (s : S), add (f s)
`ρ {n}`		multiplication by n

Metadata: Number Systems

Our generic construction needs to know how to interpret a number system; `value : Bin → Nat` isn't enough:

- A function can't open up another function definition.
- (We don't have the time to try infinitely many values.)

A simple scheme of annotations, `Number`, describes a generalization of (dense) positional number systems:

`𝟙 {n}`		constantly n
`σ S {f}`	acts as	given (s : S), add (f s)
`ρ {n}`		multiplication by n

```
BinND : Desc Number ∅
BinND = 𝟙 {0}                -- 0b : Bin
      ∷ ρ {2} _ (𝟙 {1})  -- 1b : Bin → Bin
      ∷ ρ {2} _ (𝟙 {2})  -- 2b : Bin → Bin
```

New Ornaments

New universe ⇒ new ornaments:

```
data Orn (Me' : Meta) (Δ : Tel) (re-par : Δ → Γ)
         : Desc Me Γ → Type
```

New Ornaments

New universe ⇒ new ornaments:

```
data Orn (Me' : Meta) (Δ : Tel) (re-par : Δ → Γ)
         : Desc Me Γ → Type
```

⇒ More preconditions in `ConOrn`, e.g.:

- Q: When can we ornament (`ρ f`) to (`ρ g`)?
- A: when `f ∘ re-par` is (pointwise) equal to `re-par ∘ g`.

New Ornaments

Using the new ornaments, random-access list become an ornament on binary numbers:

```
RandomOD : Orn _ (∅ ▷ const Type) ! BinND
RandomOD = 𝟙                                 -- Zero : Random A
         ∷ Δ par                             -- One  : A
         ( ρ (Double ∘ par) (const refl)     -- → Random (A × A)
           𝟙)                                -- → Random A
         ∷ Δ (Double ∘ par)                  -- Two  : A × A
         ( ρ (Double ∘ par) (const refl)     -- → Random (A x A)
           𝟙)                                -- → Random A
```

Generic Numerical Representations

We now have a universe in which we can

- describe number systems
- describe nested types
- do generic programming.

So, we can now generalize the constructions from before into one program

```
TreeOD : (ND : Desc Number ∅)
       → Orn _ (∅ ▷ const Type) ! ND
```

Idea:

- take a number system `ND : Desc Number ∅`
- using the `Number` annotations, insert fields such that size and value match up

$$
\begin{tikzcd}
\mathrm{TreeOD}\ \mathrm{ND} \arrow[d, "\mathrm{ornForget}"'] \arrow[dr, "\mathrm{size}"] & \\
\mathrm{ND} \arrow[r, "\mathrm{value}"'] & \mathrm{Nat}
\end{tikzcd}
$$

- $\Rightarrow$ `TreeOD` produces an ornament, so the numerical representation has the structure of the number system.

Generic Numerical Representations

The construction works by cases on the number system:

```
ND         → TreeOD ND
𝟙 {n}      ⇒ Δ (Vec n ∘ par) -- Vec n A
             𝟙               -- → Tree ND A
σ {f} S ND ⇒ σ S                                -- (s : S)
             (Δ (both Vec par (f ∘ var))  -- → Vec (f s) A
             (TreeOD ND))                 -- ...
ρ {n} _ ND ⇒ ρ (Vec n ∘ par) -- Tree ND (Vec n A)
             (TreeOD ND)     -- ...
```

where

- `Vec n A` is a list of n values of A
- `var` abbreviates the extractor for the last variable
- `both f a b x = f (a x) (b x)`

Example: random-access Lists

Let's look at `TreeOD` in action on binary numbers.

`TreeOD`:

```
BinND                 →  RandomOD
𝟙 {0} _               ⇒  Δ (Vec 0 A) (𝟙 _)
ρ {2} _ (𝟙 {1} _)     ⇒  ρ (Vec 2 ∘ par) (
                         Δ (Vec 1 A) (𝟙 _))
ρ {2} _ (𝟙 {2} _)     ⇒  ρ (Vec 2 ∘ par) (
                         Δ (Vec 2 A) (𝟙 _))
```

(Note:

- `Vec` A 0 is equivalent to having no field,
- `Vec` A 1 is equivalent to just A,
- `Vec` A 2 is equivalent to `Double` A.)

Conclusion

To summarize

- we made a universe to describe number systems and nested datatypes
- we equipped this with suitable ornaments
- and this rewards us with `ListOD`, `RandomOD`, and many other numerical representations, for free.

There are some more constructions in the paper:

- Composite types: binary finger trees.
- `Nat` → `List` → `Vec` ⇒ ND → `TreeOD` ND → `TrieOD` ND.
- Folding operation for `Desc`: generic programs for nested types.
- Why Cubical Agda does not break this construction (which it seems to at first glance).

Future Work

But there is still more to be explored:

- Index/path types don't have "nice" descriptions (hence no generic `lookup`) $\Rightarrow$ sigma-descriptions.
- $\Rightarrow$ Representability of `TrieOD` has no "nice" proof (yet)
- The approach can be adapted to datastructures that use branching instead of nesting (e.g, Braun trees),
- and also to sparse number systems.
- ...

Any questions?