Ornamenting Inductive-Recursive Definitions

Peio Borthelle, Conor McBride

August 28, 2018

Abstract

Contents

1 Introduction

A Technical Preliminary This research development has been exclusively done formally, using the dependently–typed language Agda ([3]) as an interactive theorem–prover. As such this report is full of code snippets, following the methodology of literate programming ([1]). Theorems are presented as type declaration, proofs are implementations of such declarations and definitions are usually some kind of datastructure introduction: it definitely lies on the *program* side of the Curry–Howard correspondance. The syntax and concepts of Agda should not be too alien to a Haskell or Coq programmer but it might be interesting to start out by reading the appendix A which presents its most important features.

Motivations Although they were probably first intended as theorem provers, dependently–typed languages are currently evolving into general–purpose programming languages, leveraging their expressivity to enable correct–by–construction type–driven programming. But without the right tools this new power is unmanageable. One issues is the need to prove over and over again the same properties for similar datastructures. Ornaments (**TODO**:*ref mcbride*) tackle this problem by giving a formal syntax to describe how datastructures might be *similar*. Using these objects, we can prove generic theorems once and for all. The broad idea behind this approach is to “speak in a more intelligible way to the computer”: if instead of giving a concrete declarations we gave defining properties, we would be able to systematically collect free theorems which hold by (some high level) definition.

The present work aims to generalize ornaments to the widest possible notion of datatypes: inductive–recursive families (or indexed inductive–recursive types) as recently axiomatized by Ghani et al (??).

Related Work

Acknowledgements This 3 month internship research project was conducted in the Mathematically Structured Programming group of the University of Strathclyde, Glasgow under supervision of Conor McBride as part of my M1 in theoretical computer science at the university of ENS de Lyon. I spend an enjoyable time there with the staff, PhD students and fellow interns, discovering a whole new world populated by modalities, coinduction, quantitative types and cheering against England. Many thanks to Ioan and Simone for sharing their roof. Last but not least I’m grateful to Conor for sharing his insights on (protestant integrist) type theory, taking the time to lead me through narrow difficulties or open doors into new realms of thought. It was loads of fun and I’m looking forward to collaborate again in some way or another.

2 Indexed Induction–Recursion

The motivation behind indexed induction–recursion is to provide a single rule that can be specialized to create most of the types that are encountered in Martin Loef’s Intuitionistic Type Theory (ITT) such as inductive types (W–types), inductive families *etc.* This rule has been inspired to Dybjer (**TODO**:*ref*) by Martin Loef’s definition of a universe à–la–Tarski, an inductive set of codes `data U : Set` and a recursive function `el : U → Set` reflecting codes into actual sets (here a simple version with only natural numbers and Π–types).

```
data U where
  `ℕ : U                                   el `ℕ       = ℕ
  `Π : (A : U) (B : el A → U) → U           el (`Π A B) = (a : el A) → el (B a)
```

We can see the most important caracteristic of inductive-recursive definitions: the simultaneous definition of an inductive type and a recursive function on it with the ability to use the recursive function in the type of the constructors, even in negative positions (left of an arrow). *Indexed* inductive-recursive definitions are a slight generalization, similar to the relationship between inductive types and inductive families. In its full generality, indexed induction recursion allows to simultaneously define an inductive predicate $\mathsf{U} : \mathsf{I} \to \mathsf{Set}$ and an indexed recursive function $\mathsf{f} : (\mathsf{i} : \mathsf{I}) \to \mathsf{U}\ \mathsf{i} \to \mathsf{X}\ \mathsf{i}$ for any $\mathsf{I} : \mathsf{Set}$ and $\mathsf{X} : \mathsf{I} \to \mathsf{Set}_1$. Using a vocabulary influenced by the *bidirectional* paradigm for typing (**TODO**:*ref*) we will call $\mathsf{i} : \mathsf{I}$ the *input index* and $\mathsf{X}\ \mathsf{i}$ the *output index*. Indeed if we think of the judgement $\mathsf{a} : \mathsf{U}\ \mathsf{i}$ as a typechecker would, the judgment requires the validity of $\mathsf{i} : \mathsf{I}$ and suffices to demonstrate the validity of $\mathsf{f}\ \mathsf{a} : \mathsf{X}\ \mathsf{i}$. We will explore bidirectionality further in section **??**.

Induction-recursion is arguably the most powerful set former (currently known) for ITT. **TODO**:*who?* has shown that its addition gives ITT a proof-theoretic strength slightly greater than KPM, Kripke–Platek set theory together with a recursive Mahlo universe. Although its proof-theoretic strength is greater than Γ_0, ITT with induction–recursion is still considered predicative in a looser constructivist sense: it arguably has bottom–to–top construction.

2.1 Categories

Since we will use category theory as our main language we first recall the definition of a category C:

- a collection of objects $\mathsf{C} : \mathsf{Set}$
- a collection of morphisms (or arrows) $\_\Rightarrow\_ : (\mathsf{X}\ \mathsf{Y} : \mathsf{C}) \to \mathsf{Set}$
- an identity $1 : (\mathsf{X} : \mathsf{C}) \to \mathsf{X} \Rightarrow \mathsf{X}$
- a composition operation $\_\circ\_ : \forall \{\mathsf{X}\ \mathsf{Y}\ \mathsf{Z}\} \to \mathsf{Y} \Rightarrow \mathsf{Z} \to \mathsf{X} \Rightarrow \mathsf{Y} \to \mathsf{X} \Rightarrow \mathsf{Z}$ that is associative and respects the identity laws $1\ \mathsf{X} \circ \mathsf{F} \equiv \mathsf{F} \equiv \mathsf{F} \circ 1\ \mathsf{Y}$

A functor F between categories C and D is a mapping of objects $\mathsf{F} : \mathsf{C} \to \mathsf{D}$ and a mapping of arrows $\mathsf{F}[\_] : \forall \{\mathsf{X}\ \mathsf{Y}\} \to \mathsf{X} \Rightarrow \mathsf{F} \to \mathsf{F}\ \mathsf{X} \Rightarrow \mathsf{F}\ \mathsf{Y}$.

2.2 Data types

The different notions of data types, by which we mean inductive types, inductive–recursive types and their indexed variants, share their semantics: initial algebras of endofunctors. In a first approximation, we can think of an "initial algebra" as the categorical notion for the "least closed set" (just not only for sets). As such, we will study a certain class of functors with initial algebras that give rise to our indexed inductive–recursive types.

We shall determine the category our data types live in. The most simple data types, inductive types, live in the category Set. On the other hand, as we have seen, inductive–recursive data types are formed by couples in $(\mathsf{U} : \mathsf{Set}) \times (\mathsf{U} \to \mathsf{X})$. Categorically, this an X-indexed set and it is an object of the slice category of Set/X. We will be representing these objects by the record type $\mathsf{Fam}\ \gamma\ \mathsf{X}$[1].

```
record Fam (α : Level) (X : Set β) : Set (lsuc α ⊔ β) where
  constructor _,_
  field
```

[1] See section **TODO**:*ref* for some explainations of Level, but for most part it can be safely ignored, together with its artifacts Lift, lift and the greek letters α, β, γ and δ.

$$\begin{aligned}
&\mathsf{Code} : \mathsf{Set}\ \alpha \\
&\mathsf{decode} : \mathsf{Code} \longrightarrow X
\end{aligned}$$

$$\begin{aligned}
&_\rightsquigarrow_ : \mathsf{Fam}\ \alpha_0\ X \longrightarrow \mathsf{Fam}\ \alpha_1\ X \longrightarrow \mathsf{Set}\ _ \\
&F \rightsquigarrow G = (i : \mathsf{Code}\ F) \longrightarrow \Sigma\ (\mathsf{Code}\ G)\ \lambda\ j \longrightarrow \mathsf{decode}\ G\ j \equiv \mathsf{decode}\ F\ i
\end{aligned}$$

This definition already gives us enough to express our first example of inductive–recursive definition.

$$\begin{aligned}
&\Pi\mathbb{N}\text{–univ} : \mathsf{Fam}\ \mathsf{lzero}\ \mathsf{Set} \\
&\Pi\mathbb{N}\text{–univ} = U , el
\end{aligned}$$

Now we can get to indexed inductive–recursive data types which essentially are functions from an input index $i : I$ to $(X\ i)$-indexed sets. We will use couples (I , X) a lot as they define the input and output indexing sets so we call their type ISet.

$$\begin{aligned}
&\mathsf{ISet} : (\alpha\ \beta : \mathsf{Level}) \longrightarrow \mathsf{Set}\ _ \\
&\mathsf{ISet}\ \alpha\ \beta = \mathsf{Fam}\ \alpha\ (\mathsf{Set}\ \beta)
\end{aligned}$$

$$\begin{aligned}
&\mathbb{F} : (\gamma : \mathsf{Level}) \longrightarrow \mathsf{ISet}\ \alpha\ \beta \longrightarrow \mathsf{Set}\ _ \\
&\mathbb{F}\ \gamma\ (I , X) = (i : I) \longrightarrow \mathsf{Fam}\ \gamma\ (X\ i)
\end{aligned}$$

$$\begin{aligned}
&_\Rightarrow_ : \mathbb{F}\ \gamma_0\ X \longrightarrow \mathbb{F}\ \gamma_1\ X \longrightarrow \mathsf{Set}\ _ \\
&F \Rightarrow G = (i : _) \longrightarrow F\ i \rightsquigarrow G\ i
\end{aligned}$$

Again we might consider our universe example as a trivially indexed type.

$$\begin{aligned}
&\Pi\mathbb{N}\text{–univ}_i : \mathbb{F}\ \mathsf{lzero}\ (\top , \lambda\ _ \longrightarrow \mathsf{Set}) \\
&\Pi\mathbb{N}\text{–univ}_i\ _ = U , el
\end{aligned}$$

TODO:*mention* FΣ *and* FΠ

2.3 A Universe of Strictly Positive Functors

Dybjer and Setzer have first presented codes for (indexed) inductive-recursive definitions (**TODO**:*ref*) by constructing a universe of functors. However, as conjectured by [2], this universe lacks closure under composition, *eg* if given the codes of two functors, we don't know how to construct a code for the composition of the functors. I will thus use an alternative universe construction devised by McBride which we call *Irish* induction–recursion[2].

In this section we fix a given pair of input/output indexes $X\ Y : \mathsf{ISet}\ \alpha\ \beta$ and i will define codes $\rho : \mathsf{IIR}\ \delta\ X\ Y : \mathsf{Set}$ for some functors $[\![\ \rho\ ]\!] : \mathbb{F}\ \gamma\ X \longrightarrow \mathbb{F}\ (\gamma \sqcup \delta)\ Y$.

First we give a datatype of codes that will describe the first component inductive–recursive functors. This definition is itself inductive–recursive: we define a type $\mathsf{poly}\ \gamma\ X : \mathsf{Set}$ representing the shape of the constructor[3] and a recursive predicate $\mathsf{info} : \mathsf{poly}\ \gamma\ X \longrightarrow \mathsf{Set}$ representing the information contained in the final datatype, underapproximating the information contained in a subnode by the output index $X\ i$ it delivers.

Lets give some intuition for these constructors.

[2] It has also been called *polynomial* induction–recursion because it draws similarities to polynomial functors, yet they are different notions and should not be confused.

[3] It is easy to show that in a dependent theory, restricting every type to a single constructor does not lose generality.

- ι i codes an inductive position with input index i, *eg* the indexed identity functor. Its info is decode X i *eg* the output index that we will obtain from the later constructed recursive function.
- κ A codes the constant functor, with straighforward information content A.
- σ A B codes the dependent sum of a functor A and a functor family B depending on A's information.
- π A B codes the dependent product, but strict positivity rules out inductive positions in the domain. As such the functor A must be a constant functor and we can (and must) make it range over Set, not poly.

The encoding of our ΠℕN–universe goes as follows:

```
data Πℕ-tag : Set where ‘ℕ ‘Π : Πℕ-tag

Πℕ0 : poly lzero (⊤ , λ _ → Set)
Πℕ0 = σ (κ Πℕ-tag) λ {   -- select a constructor
    -- no argument for ‘ℕ
  (lift ‘ℕ) → κ ⊤;
    -- first argument, an inductive position whose output index we bind to A
    -- second argument, a (non-dependent) Π type from A to an inductive position
  (lift ‘Π) → σ (ι ∗) λ {(lift A) → π A λ _ → ι ∗}}
```

We can now give the interpretation of a code ρ : poly δ X into a functor $⟦ ρ ⟧_0$.

```
⟦_⟧0 : (ρ : poly γ X) → 𝔽 δ X → Fam (γ ⊔ δ) (info ρ)
⟦_⟧0 (ι i) F = lift ◃ lft γ F i
⟦_⟧0 (κ A) F = Lift δ A , lift ∘ lower
⟦ σ A B ⟧0 F = FΣ (⟦ A ⟧0 F) λ a → ⟦ B a ⟧0 F
⟦ π A B ⟧0 F = FΠ A λ a → ⟦ B a ⟧0 F

⟦_⟧[_]0 : (ρ : poly γ X) → F ⇒ G → ⟦ ρ ⟧0 F ⇝ ⟦ ρ ⟧0 G
⟦ ι i   ⟧[ φ ]0 = λ x → let j , p = φ i $ lower x in lift j , cong lift p
⟦ κ A   ⟧[ φ ]0 = λ a → lift $ lower a , refl
⟦ σ A B ⟧[ φ ]0 = FΣ⇝ ⟦ A ⟧[ φ ]0 λ a → ⟦ B $ decode (⟦ A ⟧0 _) a ⟧[ φ ]0
⟦ π A B ⟧[ φ ]0 = FΠ⇝ λ a → ⟦ B a ⟧[ φ ]0
```

It would be time to check if this interpretation does the right thing on our example, alas even simple examples of induction–recursion are somewhat complicated, as such I don't think it would be informative to display here the normalized expression of $⟦$ ΠℕNc $⟧_0$ F. The reader is still encouraged to normalize it by hand to familiarize with the interpretation.

While taking as parameter a indexed family 𝔽 γ X, our intepreted functors only output a family Fam (γ ⊔ δ) (info ρ). In other words, ρ : poly γ X only gives the structure of the definition for a given input index i : Code Y. To account for that, the full description of the first component of inductive–recursive functors has to be a function node : Code Y → poly γ X. We are left to describe the recursive function, which can be done with a direct emit : (j : Code Y) → info (node j) → decode Y j computing the output index from the full information.

```
record IIR (γ : Level) (X Y : ISet α β) : Set (lsuc α ⊔ β ⊔ lsuc γ) where
  constructor _,_
  field
    node : (j : Code Y) → poly γ X
    emit : (j : Code Y) → info (node j) → decode Y j
```

We can now explain the index emitting function el, completing our encoding of the ΠℕN–universe.

```
ΠℕNc : IIR lzero (⊤ , λ _ → Set) (⊤ , λ _ → Set)
node ΠℕNc _ = ΠℕN₀
emit ΠℕNc _ (lift ‘ℕ , lift ∗) = ℕ
emit ΠℕNc _ (lift ‘Π , A , B) = (a : lower A) → lower $ B a

⟦_⟧ : IIR γ X Y → 𝔽 δ X → 𝔽 (γ ⊔ δ) Y
⟦ ρ ⟧ F = λ j → emit ρ j ◃ ⟦ node ρ j ⟧₀ F

⟦_⟧[_] : (ρ : IIR γ X Y) → F ⇒ G → ⟦ ρ ⟧ F ⇒ ⟦ ρ ⟧ G
⟦ ρ ⟧[ φ ] j = emit ρ j ◂ ⟦ node ρ j ⟧[ φ ]₀
```

We have use the post–composition functor defined as follows:

```
_◃_ : (f : X → Y) → Fam α X → Fam α Y
f ◃ F = _ , f ∘ decode F

_◂_ : (f : X → Y) → A ⇝ B → f ◃ A ⇝ f ◃ B
(f ◂ m) i = let (j , p) = m i in j , cong f p
```

2.4 Initial Algebra

2.4.1 Least Fixed–Point

Now that we have a universe of functors, we need to translate that into actual indexed inductive–recursive types. This amounts to taking its least fixed–point μ ρ.

```
μ : (ρ : IIR γ X X) → 𝔽 γ X
Code (μ ρ i) = μ-c ρ i
decode (μ ρ i) = μ-d ρ i
```

It consists of two parts, the inductive family μ-c ρ : Code X → Set and the recursive function μ-d ρ : (i : Code X) → μ-c ρ i → decode X i. By chance Agda has a primitive for constructing these kinds of sets: the **data** keyword. Applying the interpreted functor to the least fixed–point with ⟦ ρ ⟧ (μ ρ) and the two components of the indexed family basically gives us the implementation of respectively μ-c ρ and μ-d ρ.

```
data μ-c (ρ : IIR γ X X) (i : Code X) : Set γ where
  ⟨_⟩ : Code (⟦ ρ ⟧ (μ ρ) i) → μ-c ρ i
μ-d : (ρ : IIR γ X X) (i : Code X) → μ-c ρ i → decode X i
μ-d ρ i ⟨ x ⟩ = decode (⟦ ρ ⟧ (μ ρ) i) x
```

We have now completed the encoding of ΠℕN and we can write pretty versions the constructors!
TODO:*minipage*

```
U₁ : Set
U₁ = μ-c ΠℕNc ∗
el₁ : U₁ → Set
el₁ = μ-d ΠℕNc ∗
‘ℕ₁ : U₁
```

```
`ℕ₁ = ⟨ lift `ℕ , lift * ⟩
`Π₁ : (A : U₁) (B : el₁ A → U₁) → U₁
`Π₁ A B = ⟨ lift `Π , lift A , lift ∘ B ⟩
```

2.4.2 Catamorphism and Paramorphism

I previously said that this least–fixed point has in category theory the semantic of an initial algebra. Let's break it down. Given an endofunctor F : C → C, an F-algebras is a carrier X : C together with an arrow F X ⇒ X. An arrow between two F-algebras (X , φ) and (Y , ψ) is an arrow m : X ⇒ Y subject to the commutativity of the usual square diagram ψ ∘ F[m] ≡ m ∘ φ.

$$\begin{tikzcd} F\,X \arrow[r, "\varphi"] \arrow[d, "F[\,m\,]"'] & X \arrow[d, "m"] \\ F\,Y \arrow[r, "\psi"] & Y \end{tikzcd}$$

Additionaly, an object X : C is initial if for any Y : C we can give an arrow X ⇒ Y.

We almost already have constructed an ⟦ ρ ⟧-algebra with carrier μ ρ and the constructor ⟨_⟩ mapping the object part of ⟦ ρ ⟧ (μ ρ) to μ ρ. What is left is to add a trivial proof.

```
roll : ⟦ ρ ⟧ (μ ρ) ⇒ μ ρ
roll _ x = ⟨ x ⟩ , refl
```

TODO:*interlude: intro example distinct elt list*

To prove the fact that our algebra is initial we have first have to formally write the type of algebras.

```
record alg (δ : Level) (ρ : IIR γ X X) : Set (α ⊔ β ⊔ lsuc δ ⊔ γ) where
  constructor _,_
  field
    {obj} : 𝔽 δ X
    mor : ⟦ ρ ⟧ obj ⇒ obj
open alg public
```

We can now give for every φ : alg δ ρ the initiality arrow μ ρ ⇒ obj φ.

```
fold : (φ : alg δ ρ) → μ ρ ⇒ obj φ
fold φ = mor φ ⊙ foldm φ
```

With the helper foldm ρ is defined as:

```
foldm : (φ : alg δ ρ) → μ ρ ⇒ ⟦ ρ ⟧ (obj φ)
foldm {ρ = ρ} φ i ⟨ x ⟩ = ⟦ ρ ⟧[ fold φ ] i x
```

Complying to the proof obligation for the equality condition, we get:

```
foldm–⊙ : (φ : alg δ ρ) → foldm φ ⊙ roll ≡ ⟦ ρ ⟧[ fold φ ]
foldm–⊙ φ = funext λ i → funext λ x → cong–Σ refl (uoip _ _)

fold–⊙ : (φ : alg δ ρ) → fold φ ⊙ roll ≡ mor φ ⊙ ⟦ ρ ⟧[ fold φ ]
fold–⊙ φ = trans ⊙–assoc $ cong (_⊙_ $ mor φ) (foldm–⊙ φ)
```

Note that we make use of uoip the unicity of identity proofs, together with the associativity lemma ⊙–assoc.

As hinted by its name, the initiality arrow fold ρ is in fact a generic fold or with fancier wording an elimination rule, precisely the catamorphism (also called recursor). An elimination scheme is the semantic of recursive functions with pattern matching. Diggressing a little on elimination rules, we can notice that this is not the only one.

TODO:*introduce paramorphism, factorial on nat* **TODO**:*para is the most generic (non-dependent) eliminator, ref meeertens*

```
record alg≈ (δ : Level) (Y : Code X → Set β₁) (ρ : IIR γ X X) : Set (α ⊔ β₀ ⊔ β₁ ⊔ lsuc δ ⊔ γ) where
  constructor _,_
  field
    {obj} : 𝔽 δ (Code X , Y)
    down : (i : Code X) → decode X i → Y i
    mor : (down ⊲ ⟦ ρ ⟧ (μ ρ & obj)) ⇒ obj
open alg≈ public

para₀ : (Y : Code X → Set β′) (φ : alg≈ δ Y ρ) → μ ρ ⇒ μ ρ & obj φ
π₀ (para₀ Y φ i ⟨ x ⟩) = ⟨ x ⟩ , π₀ $ mor φ i (π₀ $ ⟦ ρ ⟧[ para₀ Y φ ] i x)
π₁ (para₀ Y φ i ⟨ x ⟩) = refl

para : (Y : Code X → Set β′) (φ : alg≈ δ Y ρ) → (down φ ⊲ μ ρ) ⇒ obj φ
π₀ (para Y φ i ⟨ x ⟩) = π₀ $ mor φ i (π₀ $ ⟦ ρ ⟧[ para₀ Y φ ] i x)
π₁ (para Y φ i ⟨ x ⟩) = trans (π₁ $ mor φ i _) (cong (down φ i) (π₁ $ ⟦ ρ ⟧[ _ ] i x))
```

2.5 Induction Principle

We have given several elimination rules, but dependent languages are used to do mathematics and the only elimination rule a mathematican would want on an inductive type is the most powerful one: an induction principle. In substance the induction principle states that, for any predicate P : (i : Code X) (x : Code (μ ρ i)) → Set, if given that the predicate holds for every subnode we can show it hold for the node itself, then we can show the predicate to hold for every possible node.

Let's formalize that a bit. I define a predicate all stating that a property hold for all subnodes. It looks a lot like ⟦ ρ ⟧ but does something slightly more powerful at inductive positions.

```
all : (ρ : poly γ X) (P : ∀ i → Code (F i) → Set δ) →
      Code (⟦ ρ ⟧₀ F) → Set (α ⊔ γ ⊔ δ)
all (ι i)     P (lift x) = Lift (α ⊔ γ) (P i x)
all (κ A)     P x        = ⊤
all (σ A B) P (a , b) = Σ (all A P a) λ _ → all (B (decode (⟦ A ⟧₀ _) a)) P b
all (π A B) P f          = (a : A) → all (B a) P (f a)
```

Given that I can state the induction principle.

```
induction : (ρ : IIR γ X X) (P : ∀ i → Code (μ ρ i) → Set δ)
  (p : ∀ i (xs : Code (⟦ ρ ⟧ (μ ρ) i)) → all (node ρ i) P xs → P i (⟨_⟩ xs)) →
  (i : Code X) (x : Code (μ ρ i)) → P i x
induction ρ P p i ⟨ x ⟩ = p i x $ every (node ρ _) P (induction ρ P p) x
```

I used the helper every which explains how to construct a proof of all for ⟦ ρ ⟧ F if we can prove the predicate for F.

```
every : (ρ : poly γ X) (P : ∀ i → Code (D i) → Set δ)
```

```
        (p : ∀ i (x : Code (D i)) → P i x) (xs : Code (⟦ ρ ⟧₀ D)) →
        all ρ P xs
every (ι i)        _ p (lift x)  =  lift $ p i x
every (κ A)     P _ _           =  ⋆
every (σ A B) P p (a , b)  =  every A P p a , every (B (decode (⟦ A ⟧₀ _) a)) P p b
every (π A B) P p f          =  λ a → every (B a) P p (f a)
```

Note that I could have derived the other elimination rules from this induction principle, but cata– and paramorphisms are very useful non–dependent special cases that diserve to be treated separately and possibly optimized. Non-dependent functions still have a place of choice in dependent languages: just because we can replace every implication by universal quantification doesn't mean we should.

3 Ornaments

3.1 Fancy Data

A major use for indexes in type families is to refine a type to contain computational relevant information about objects of that type. Suppose we have a type of lists.

```
data list (X : Set) : Set where
  nil  : list X
  cons : X → list X → list X
```

We may want to define a function zip : list X → list Y → list (X × Y) pairing up the items of two arguments.

```
zip : list X → list Y → list (X × Y)
zip nil            nil              =  nil
zip (cons x xs) (cons y ys)  =  cons (x , y) (zip xs ys)
zip (cons x xs) nil            =  ?
zip nil            (cons y ys)  =  ?
```

It is clear that there is nothing really sensible to do for the two last cases. We should signal some incompatibility by throwing an exception or we may just return an empty list. But this is not very principled. What we would like is to enforce on the type level that the two arguments have the same length and that we additionally will return a list of that exact length. This type is called vec.

```
data vec (X : Set) : ℕ → Set where
  nil  : vec X zero
  cons : ∀ {n} → X → vec X n → vec X (suc n)
```

I wrote the constructors such that they maintain the invariant that vec X n is only inhabited by sequences of length n. I may now write the stronger version of zip which explicitely states what is possible to zip.

```
zip : {X Y : Set} {n : ℕ} → vec X n → vec Y n → vec (X × Y) n
zip nil            nil              =  nil
zip (cons x xs) (cons y ys)  =  cons (x , y) (zip xs ys)
```

This is made possible because of the power dependent pattern matching has: knowing a value is of a particular constructor may add constraints to the type of the expression we have to produce and to the type of other arguments. As such when we pattern match with cons on the first argument, the implicit index n gets unified with suc m, which implies that the second argument has no choice but to be a cons too.

Several comments can be made about vec and list. The first one is that they are almost same. More precisely, they have the same shape, the only added argument is the natural number n in cons for vec[4]. Because only a sprinkle of information has been added to something of the same shape, we should be able to derive a function from vec X n to list X. The second comment is that there is an straightforward isomorphism between list X and Σ ℕ (vec X). As such we should be able to come up with the reverse function (x : list X) → vec X (length x).

The rest of this section will be dedicated to formalizing prose definitions such as "vectors are lists indexed by their length" and generically deriving the properties that they imply.

3.2 Reindexing

Another take on the previous example of lists and vectors is that vectors have a more informative index (natural numbers) than lists (trivial indexation by the unit type). This can be expressed by the fact that there is a function ℕ → ⊤ giving a non-fancy index given a fancy one. Because we work with inductive–recursive types and not just inductive ones, we have two indexes—the input index I : Set and the output index X : I → Set—and we have to translate this notion. For this we introduce the datatype PRef (index refinement using powersets).

```
record PRef (α₁ β₁ : Level) (X : ISet α₀ β₀) : Set (α₀ ⊔ β₀ ⊔ lsuc α₁ ⊔ lsuc β₁) where
  field
    Code : Set α₁
    down : Code → Fam.Code X
    decode : (j : Code) → decode X (down j) → Set β₁
open PRef public
```

Let X : ISet α_0 β_0 and R : PRef α_1 β_1 X. Code R represents the new input index, together with the striping function down R taking new input indexes to old ones. Additionaly we have to define a new output index Y : Code R → Set such that we can derive a stripping function (j : Code R) → Y j → X (down j). Directly defining Y together with this second striping function would not have been practical[5]. Thus instead of the stripping function, we ask for its fibers (called its graph), given by decode R. This reversal is the classical choice between families (A : Set) × A → X and powersets X → Set to represent indexation.

Because of the small fiber twist we operated, we have a bit of work to get the new indexing pair (in ISet) from a PRef.

```
PFam : PRef α₁ β₁ X → ISet α₁ (β₀ ⊔ β₁)
Code (PFam P) = Code P
decode (PFam P) j = Σ _ (decode P j)
```

In substance, the new output index is simply the old one to which we add some information that can depend on it. The stripping function is thus simply the projection π_0.

[4]Actually this n does not contain any information as it can be derived from the type index. As such there is ongoing research to optimize away these kind of arguments and we will see that because of our index–first formalism of indexed datatypes it will not even be added in the first hand.

[5]Later we would have needed to define preimages which necessarily embed some notion of equality. As explained in ?? we want to avoid any mention of equality when formalizing the unrelated matters of data types.

3.3 A Universe of Ornaments

Step by step, following the construction of induction–recursion I will start by describing ornaments of poly, the inductive part of the definition. For $R : \mathsf{PRef}\ \alpha_1\ \beta_1\ X$ and $\rho : \mathsf{poly}\ \gamma_0\ X$ I define a universe of decriptions $\mathsf{orn}_0\ \gamma\ R\ \rho : \mathsf{Set}\ \_$. Simultaneously I define an interpretation $\lfloor\ o\ \rfloor_0 : \mathsf{poly}\ (\gamma_0 \sqcup \gamma_1)\ (\mathsf{PFam}\ R)$ taking the description of the "delta" to the actual fancy description it represents, and a stripping function $\mathsf{info}{\downarrow} : \mathsf{info}\ \lfloor\ o\ \rfloor_0 \longrightarrow \mathsf{info}\ \rho$ taking new node informations to old ones.

```
data orn₀ (γ₁ : Level) (R : PRef α₁ β₁ X) : poly γ₀ X → Set
⌊_⌋₀ : (o : orn₀ γ₁ R ρ) → poly (γ₀ ⊔ γ₁) (PFam R)
info↓ : info ⌊ o ⌋₀ → info ρ

data orn₀ γ₁ R where
  ι     : (j : Code R)                                    → orn₀ γ₁ R (ι (down R j))
  κ     : {A : Set γ₀}                                    → orn₀ γ₁ R (κ A)
  σ     : (A : orn₀ γ₁ R U)
          (B : (a : info ⌊ A ⌋₀) → orn₀ γ₁ R (V (info↓ a)))
            → orn₀ γ₁ R (σ U V)
  π     : (B : (a : A) → orn₀ γ₁ R (V a)) → orn₀ γ₁ R (π A V)
  add₀  : (A : poly (γ₀ ⊔ γ₁) (PFam R))
          (B : info A → orn₀ γ₁ R U)                      → orn₀ γ₁ R U
  add₁  : (A : orn₀ γ₁ R U)
          (B : info ⌊ A ⌋₀ → poly (γ₀ ⊔ γ₁) (PFam R)) → orn₀ γ₁ R U
  del-κ : (a : A) → orn₀ γ₁ R (κ A)

⌊ ι j        ⌋₀ = ι j
⌊_⌋₀ (κ {A}) = κ (Lift γ₁ A)
⌊ σ A B      ⌋₀ = σ ⌊ A ⌋₀ λ a → ⌊ B a ⌋₀
⌊_⌋₀ (π {A} B) = π (Lift γ₁ A) λ {(lift a) → ⌊ B a ⌋₀}
⌊ add₀ A B ⌋₀ = σ A λ a → ⌊ B a ⌋₀
⌊ add₁ A B ⌋₀ = σ ⌊ A ⌋₀ B
⌊ del-κ _    ⌋₀ = κ ⊤

info↓ {o = ι i}          (lift (x , _)) = lift x
info↓ {o = κ}            (lift a)       = lift $ lower a
info↓ {o = σ A B}        (a , b)        = info↓ a , info↓ b
info↓ {o = π B}          f              = λ a → info↓ (f $ lift a)
info↓ {o = add₀ A B} (a , b)        = info↓ b
info↓ {o = add₁ A B} (a , _)        = info↓ a
info↓ {o = del-κ a}      _              = lift a
```

Lets break down the constructors. First we have the constructors that look like poly: ι, κ, σ and π. They essentially say that nothing is changed. $\iota\ j$ ornaments poly of the form $\iota\ i$ where $\mathsf{down}\ R\ j \equiv i$ *ie* we replace inductive positions by a fancy index such that the stripping matches. $\sigma\ A\ B$ has to use the interpretation $\lfloor\_\rfloor_0$ and $\mathsf{info}{\downarrow}$ to express how the family B depends on the info of A. κ and $\sigma\ B$ change nothing and as such some of their arguments are implicit because there is no choice possible.

The next 3 constructors allow to change things. add_0 allows to delay the ornamenting, it interprets into a σ where the first component has no counterpart in the initial data. In other words we add a new argument to the constructor and then give an ornament which might depend on it. add_1 is the other way around, it gives an ornament and then adds new arguments which might depend on it. And finally del–κ allows you to erase a constant argument given that you can provide an element of it. It is restricted to delete only constants because for an inductive position it is not really clear what the notion of "element of it" is.

⌊_⌋₀ and info↓ are straightforward, the first 4 constructors are unsurprising, the additions interpret into sigmas where info↓ ignores the new component and the deletion interprets into a trivial constant, info↓ giving back the element we have provided in the definition.

As for inductive–recursive types in this part of the construction we are not yet taking input indexes into account so we can't give the ornament of lists into vectors yet. But we can give the ornament of natural numbers into lists: we identify zero with nil and suc with cons where cons demands an additional constant argument in addition to the recursive position.

```
data ℕ–tag : Set where `ze `su : ℕ–tag

nat–c : poly lzero (⊤ , λ _ → ⊤)
nat–c = σ (κ ℕ–tag) λ {
  (lift `ze) → κ ⊤;
  (lift `su) → ι ∗ }

list–R : PRef lzero lzero (⊤ , λ _ → ⊤)
Code list–R = ⊤
down list–R _ = ∗
decode list–R _ _ = ⊤

list–o : (X : Set) → orn₀ lzero list–R nat–c
list–o X = σ κ λ {
  (lift (lift `ze)) → κ                  ;
  (lift (lift `su)) → add₀ (κ X) λ _ → ι ∗}
```

I define the type orn γ_1 R S ρ : Set ornamenting ρ : IIR γ_0 X Y.

```
record orn (γ₁ : Level) (R : PRef α₁ β₁ X) (S : PRef α₁ β₁ Y) (ρ : IIR γ₀ X Y) : Set where
  field
    node : (j : Code S) → orn₀ γ₁ R (node ρ (down S j))
    emit : (j : Code S) → (x : info ⌊ node j ⌋₀) → decode S j (emit ρ (down S j) (info↓ x))
```

node is not surprising, for every fancy input index we give an ornament of the description with the corresponding old index. The emit function starts off like the one for IIR, taking an input index and the info, here of the interpretation of the ornament. Having that, we can already compute the old decoding using info↓ and emit ρ (down R j). We thus require to generate an output index compatible with the old output index we have derived.

We eventually reach the full interpretation ⌊_⌋ taking an ornament to a fancy IIR.

```
⌊_⌋ : (o : orn γ₁ R S ρ) → IIR (γ₀ ⊔ γ₁) (PFam R) (PFam S)
node ⌊ o ⌋ j = ⌊ node o j ⌋₀
emit ⌊ o ⌋ j = λ x → _ , emit o j x
```

TODO:*list to vec here?*

3.4 Ornamental Algebra

Recalling the first remark we made on the relation between an ornamented data type and it's original version, we want to generically derive an arrow mapping the new fancy one to the old one. Note that I did write arrow and not simply function: because we work in the category of indexed type families we don't simply want a map from new inductive nodes to old ones, we want it to assign output indexes consistently with the reindexing. The function we want to write has the following type.

$$\mathrm{forget} \;:\; (o \;:\; \mathrm{orn}\ \gamma_1\ R\ R\ \rho)\ \{s\} \;\longrightarrow\; \pi_0 < (\mu \lfloor o \rfloor) \Rightarrow (\mu\ \rho \circ \mathrm{down}\ R)$$
$$\mathrm{forget} \;=\; \boxed{?}$$

Because of some complications I didn't manage to implement it, but I am convinced that the missing parts are not very consequent. Indeed for inductive types, the proof is done by a fold, on the ornamental algebra $[\![\lfloor o \rfloor]\!]\ (F \circ \mathrm{down}\ R) \Rightarrow ([\![\rho]\!]\ F \circ \mathrm{down}\ R)$. The complication for induction–recursion is that this arrow cannot exist since because of the output index the two objects do not live in the same category and $F \circ \mathrm{down}\ R$ is not a valid argument to $[\![\lfloor o \rfloor]\!]$.

Some analysis has shown that in fact fold is not powerful enough to express forget and we need to resort to a paramorphism. To provide some intuition lets break down forget. It has to turn an instance of a fancy datatype into the base one. Naturally it will proceed by structural recursion, simplifying the structure bottom up. This is what the ornamental algebra $\mathrm{erase} \;:\; [\![\lfloor o \rfloor]\!]\ (F \circ \mathrm{down}\ R) \Rightarrow (\mu\ \rho \circ \mathrm{down}\ R)$ should implement: given a node where every subnode already has been simplified, simplify the current node. The reason why this halfway simplified data structure cannot exist (signified by the type mismatch of the object fed to the functor) is that this object $F \circ \mathrm{down}\ R$ does not contain enough information. In a fancy $\sigma\ A\ B$ node, A might contain inductive positions, such that the family B may depend on their (fancy) output index, something we cannot get because being a subnode, A has already been replaced by a simplified version that no longer contains this fancy output index. As such, while simplifying the datastructure, we need to keep track not only of simplified subnodes, but also of their original version, to be able to simplify the current node. This makes explicit the need for paramorphisms, which is the reason why I introduced them earlier.

Note that a finer approach would be not to resort to fully featured paramorphisms. Indeed, to simplify a node we don't need the full couple of the simplification and the fancy subnodes, we just need to reconstruct the fancy output index and we already have the simplified subnode. Thus what we exactely need is the information that is in the fancy node that isn't in the simplified one. While seemingly tortuous, this notion is very familiar and we call it a *reornament*. Indeed we have seen that lists are an ornament of natural numbers and vectors are lists indexed by natural number. Then what is a vector if it is not *all the information that is in a list but not in it's length*? This builds up a nice transition because reornaments will arise in the next subsection. This last remark that the construction of the forgetful map depends on the prior formalization of reornaments is a small funny discovery because the notion had previously been presented only afterwards. It is indeed not excluded that the two construction actually depend on each other and must be constructed simultaneously.

3.5 Algebraic Ornaments

Lets focus on the second remark we stated on the relationship between lists and vectors: the isomorphism between list and $\Sigma\ \mathbb{N}\ \mathrm{vec}$. More precisely to for each $xs \;:\; \mathrm{list}$ we can naturally associate $xs' \;:\; \mathrm{vec}\ (\mathrm{length}\ xs)$. length is no stranger, it is a very simple fold, *eg* the underlying core is an algebra $[\![\ \mathrm{list} - c\ ]\!]\ \mathbb{N} \Rightarrow \mathbb{N}$. A natural generalization follows in which for a given algebra $[\![\rho]\!]\ X \Rightarrow X$ we create an ornament indexing elements of $\mu\ \rho$ by the result of their fold. This is what we call an algebraic ornament.

In the theory of ornaments on inductive definitions there is only one index, the input index. But since we now also have an output index we might ask wether we want to algebraically ornament on the input or the output. In the case of the length algebra of lists, the input algebraic ornament gives rise to vectors, whereas the output algebraic ornament gives rise to an inductive–recursive definition where the inductive part is still list and the recursive part is the length function. As such, it seems to be a waste of power to redefine lists inductive–recursively with their length if we already separately have defined lists and the length algebra, from which we can derive length with the generic fold. We will thus only present input algebraic ornaments.

First lets define the reindexing. We suppose the indexes of our data type are X : ISet α_0 β_0 and the carrier of our algebra is F : 𝔽 α_1 X.

```
AlgR : (F : 𝔽 α₁ X) → PRef (α₀ ⊔ α₁) β₀ X
Code (AlgR F) = Σ (Code X) λ i → Code (F i)
down (AlgR F) (i , _) = i
decode (AlgR F) (i , c) x = decode (F i) c ≡ x
```

This definition simply extends the input index by an inductive element of the carrier, *eg* the specification of what output we want for the fold. Note that we also add something to the output index, namely a proof that the recursive part of the carrier matches the original output index. This is not just a *by–the–way* property, it is provable but also a crucial lemma for the construction.

As usual now I first give the pre–ornament orn$_0$ for a poly, which we will expand in a second step to full ornaments on IIR.

```
algorn₀ : (ρ : poly γ₀ X) (F : 𝔽 α₁ X) (x : Code (⟦ ρ ⟧₀ F)) →
   Σ (orn₀ (γ₀ ⊔ α₁) (AlgR F) ρ) λ o → (y : info ⌊ o ⌋₀) → decode (⟦ ρ ⟧₀ F) x ≡ info↓ y
algorn₀ (ι i) F (lift x) = ι (i , x) , λ {(lift (a , b)) → cong lift b}
algorn₀ (κ A) F (lift x) = del-κ x , λ _ → refl
algorn₀ (σ A B) F (a , b) =
   let (oa , p) = algorn₀ A F a in
   let aux x = algorn₀ (B _) F (subst (λ x → Code (⟦ B x ⟧₀ F)) (p x) b) in
   (σ oa (π₀ ∘ aux)) ,
   λ {(x , y) → cong-Σ (p x) (trans (cong₂ (λ x₁ → decode (⟦ B x₁ ⟧₀ F)) (p x)
                                              (sym $ subst-elim _ _))
                                   (π₁ (aux x) y))}
algorn₀ (π A B) F x =
   let aux a = algorn₀ (B a) F (x a) in
   π (π₀ ∘ aux) , (λ f → funext λ a → π₁ (aux a) (f $ lift a))
```

Note that the two last parts of the type are similar to an arrow between on Fam. I didn't look deeply into that but it seems like this is some sort of arrow family from ⟦ ρ ⟧$_0$ F to (orn$_0$ (γ_0 ⊔ α_1) (AlgR F) ρ , λ o → (y : info ⌊ o ⌋$_0$) → info↓ y).

More importantly, F is still the carrier of the algebra and we recursively construct an ornament whose info↓ matches with the output of ⟦ ρ ⟧$_0$ F. This ensures that we propagate good shape constraints throughout the structure, ensuring that we indeed constrain the node shapes to fold to a given target. Before proceeding with the full definition I introduce the type of fibers for a function[6].

```
data _⁻¹_ (f : X → Y) : Y → Set α where
   ok : (x : X) → f ⁻¹ (f x)
```

Now we have the building blocks for the final definition.

[6] The careful reader will be puzzled by the fact that I previously said wanting to avoid fibers and any mentionning of equality. But here there is no way around and we really want this fiber. As a consolation we can argue that this is no longer part of our *core theory of datatypes* and sidesteps are thus less consequential.

```
algorn : (ρ : IIR γ₀ X X) (φ : alg α₁ ρ) → orn (γ₀ ⊔ α₁) (AlgR (obj φ)) (AlgR (obj φ)) ρ
node (algorn ρ φ) (i , c) =
  add₀ (κ ((π₀ ∘ mor φ i) ⁻¹ c))
       λ {(lift (ok x)) → π₀ $ algorn₀ (node ρ i) (obj φ) x}
emit (algorn ρ φ) (i , c) (lift (ok x) , y) =
  trans (π₁ $ mor φ i x)
        (cong (emit ρ i) $ π₁ (algorn₀ (node ρ i) (obj φ) x) y)
```

The type is straightforward but an interesting fact is that we don't directly delegate the implementation of node to algorn_0. Indeed we have to come up with an element $x : \text{Code}\ (\llbracket \rho \rrbracket_0\ F)$. The explaination for this is that unlike our list and vector example, not every algebraic ornament has a single choice for a given index: there might still be several possible choices of constructors that will have a given fold value. We can't (and shouldn't) make that choice so we have to ask it beforehand. This choice then uniquely determines the shape of the ornament which we can unroll by a call to algorn_0. The emit part simply fulfills the proof obligation that we added in the output index.

The next step is to provide the injection from simple data into the new data indexed by the value of its fold. Again I didn't fully finish this part because the proof is tremendously hairy. The proof is done by induction, but it is completely unscrutinable because since we are working not on native Agda datatypes but on our constructed versions, we cannot use native pattern matching and recursion but have to call our generic induction principle. It's not that there is much choice on what to do, but simply that because of all the highly generic objects in use, Agda has a hard time helping us out and expanding the the right definitions just as much as we want. All in all this leads to huge theorem statements from which it is hard to tell apart the head and the tail. The beginning goes as follows.

```
algorn − inj : (i : Code X) (x : μ-c ρ {s} i) → μ-c ⌊ algorn ρ φ ⌋ (i , π₀ $ fold φ i x)
algorn − inj = induction ρ P rec
  where
    P : (i : Code X) (x : μ-c ρ {s} i) → Set _
    P i x = μ-c ⌊ algorn ρ φ ⌋ (i , π₀ $ fold φ i x)

    aux : (ρ₀ : poly γ₀ X) (x : Code (⟦ ρ₀ ⟧₀ (μ ρ))) (p : all ρ₀ P x) →
      Σ (Code (⟦ ⌊ π₀ $ algorn₀ ρ₀ (obj φ) (π₀ $ ⟦ ρ₀ ⟧[ fold φ ]₀ x) ⌋₀ ⟧₀ (μ ⌊ algorn ρ φ ⌋)))
          λ y → decode (⟦ ρ₀ ⟧₀ (μ ρ)) x
                  ≡ info↓ (decode (⟦ ⌊ π₀ $ algorn₀ ρ₀ (obj φ) (π₀ $ ⟦ ρ₀ ⟧[ fold φ ]₀ x) ⌋₀ ⟧₀
                               (μ ⌊ algorn ρ φ ⌋)) y)

    aux (ι i) (lift x) (lift p) = lift p , cong lift ?
    aux (κ A) x p = lift ∗ , refl
    aux (σ A B) (x , y) (p , q) = ?
    aux (π A B) x p =
      let aux a = aux (B a) (x a) (p a) in
      π₀ ∘ aux ∘ lower , funext (π₁ ∘ aux)

    rec : (i : Code X) (x : Code (⟦ ρ ⟧ (μ ρ) i)) → all (node ρ i) P x → P i (⟨_⟩ x)
    rec i x p =
      let c = ⟦ ρ ⟧[ fold φ ] i x in
      ⟨ lift (ok $ π₀ c) , π₀ $ aux (node ρ i) x p ⟩
```

4 Case Study: Bidirectional Simply-Typed Lambda Calculus

??

5 Discussion

5.1 Index-First Datatypes and a Principled Treatment of Equality

?? **TODO**:*bidirectional flow discipline in formalizations* **TODO**:*no choice about equality, explicit proof obligation instead of weird pattern matching conditions*

5.2 Further Work

TODO:*extend to fibred IR* **TODO**:*precise the paramorphism thing* **TODO**:*study datastructure reorganizations (eg optimizations)* **TODO**:*coalgebraic ornaments to make use of index–first*

A Introduction to Agda

B Bibliography

References

[1] Donald E. Knuth. Literate programming. *THE COMPUTER JOURNAL*, 27:97–111, 1984.

[2] Fredrik Nordvall Forsberg Neil Ghani, Conor McBride and Stephan Spahn. Variations on Inductive-Recursive Definitions. In Kim G. Larsen, Hans L. Bodlaender, and Jean-Francois Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, volume 83 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 63:1–63:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.