Ornamenting Inductive-Recursive Definitions

Peio Borthelle, Conor McBride

August 27, 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

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)
```

TODO:*alternate lines*

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 U : I ⟶ Set and an indexed recursive function f : (i : I) ⟶ U i ⟶ X i for any I : Set and X : I ⟶ Set_1. Using a vocabulary influenced by the *bidirectional* paradigm for typing (**TODO**:*ref*) we will call i : I the *input index* and X i the *output index*. Indeed if we think of the judgement a : U i as a typechecker would, the judgment requires the validity of i : I and suffices to demonstrate the validity of f a : X 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 C : Set
- a collection of morphisms (or arrows) _⇒_ : (X Y : C) ⟶ Set
- an identity 1 : (X : C) ⟶ X ⇒ X
- a composition operation _∘_ : ∀ {X Y Z} ⟶ Y ⇒ Z ⟶ X ⇒ Y ⟶ X ⇒ Z that is associative and respects the identity laws 1 X ∘ F ≡ F ≡ F ∘ 1 Y

A functor F between categories C and D is a mapping of objects F : C ⟶ D and a mapping of arrows F[_] : ∀ {X Y} ⟶ X ⇒ F ⟶ F X ⇒ F 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 endfunctors. 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 (U : Set) × (U ⟶ 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 Fam γ X[1].

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

_⇝_ : Fam α₀ X ⟶ Fam α₁ X ⟶ Set _
F ⇝ G = (i : Code F) ⟶ Σ (Code G) λ j ⟶ decode G j ≡ decode F i
```

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

```
Πℕ-univ : Fam lzero Set
Πℕ-univ = U , el
```

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.

```
ISet : (α β : Level) ⟶ Set _
ISet α β = Fam α (Set β)

𝔽 : (γ : Level) ⟶ ISet α β ⟶ Set _
𝔽 γ (I , X) = (i : I) ⟶ Fam γ (X i)
```

[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.

```
_⇒_ : 𝔽 γ₀ X ⟶ 𝔽 γ₁ X ⟶ Set _
F ⇒ G = (i : _) ⟶ F i ⇝ G i
```

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

```
Πℕ-univᵢ : 𝔽 lzero (⊤ , λ _ ⟶ Set)
Πℕ-univᵢ _ = U , el
```

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 : ISet α β and i will define codes ρ : IIR δ X Y : Set for some functors ⟦ ρ ⟧ : 𝔽 γ X ⟶ 𝔽 (γ ⊔ δ) 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 poly γ X : Set representing the shape of the constructor[3] and a recursive predicate info : poly γ X ⟶ 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.

- 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.

[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.

- 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 Πℕ–universe.

```
Πℕc : IIR lzero (⊤ , λ _ → Set) (⊤ , λ _ → Set)
node Πℕc _ = Πℕ₀
emit Πℕc _ (lift `ℕ , lift *) = ℕ
emit Πℕc _ (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 $F : C \to C$, an F-algebras is a carrier $X : C$ together with an arrow $F\ X \Rightarrow X$. An arrow between two F-algebras (X , φ) and (Y , ψ) is an arrow $m : X \Rightarrow Y$ subject to the commutativity of the usual square diagram $\psi \circ F[\ m\ ] \equiv m \circ \varphi$.

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

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

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
```

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.

3.2 A Universe of Ornaments

3.3 Ornamental Algebra

3.4

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)*

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.