Ornamenting Inductive-Recursive Definitions

Peio Borthelle, Conor McBride

August 28, 2018

Abstract We present a universe (a datatype of datatype descriptions) of endofunctors with initial algebras that give rise to indexed inductive–recursive types, *eg* simultaneous definition of an inductive type family and a recursive function on it. We provide a generic induction principle as well as some other elimination rules. Building upon that, we define a universe of *ornaments*, describing how to create an fancy version of a given datatype by enriching its indexing sets while keeping the same inductive tree shape. This allows us to introduce datatypes by giving more high–level definitions than just their description, which in turns allows to collect free theorems stated generically.

Contents

1 Introduction **2**

2 Indexed Induction–Recursion **2**

- 2.1 Categories . . . 3
- 2.2 Data types . . . 3
- 2.3 A Universe of Strictly Positive Functors . . . 4
- 2.4 Initial Algebra . . . 6
  - 2.4.1 Least Fixed–Point . . . 6
  - 2.4.2 Catamorphism and Paramorphism . . . 7
- 2.5 Induction Principle . . . 8

3 Ornaments **9**

- 3.1 Fancy Data . . . 9
- 3.2 Reindexing . . . 10
- 3.3 A Universe of Ornaments . . . 11
- 3.4 Ornamental Algebra . . . 13
- 3.5 Algebraic Ornaments . . . 14

4 Discussion **16**

- 4.1 Index-First Datatypes and a Principled Treatment of Equality . . . 16
- 4.2 Further Work . . . 17

A Bibliography **17**

B Introduction to Agda **18**

- B.1 Syntax and concepts . . . 19
- B.2 Universe Levels . . . 20
- B.3 Prelude . . . 21

C Case Study: Bidirectional Simply-Typed Lambda Calculus 21
C.1 Bidirectional Typing 21
C.2 Native Agda 22
C.3 Well–Scoped Terms 23
C.4 Well–Typed Terms 24

1 Introduction

A Technical Preliminary This research development has been exclusively done formally, using the dependently–typed language Agda (U. Norell, [13]) as an interactive theorem–prover. As such this report is full of code snippets, following the methodology of literate programming (D. Knuth, [9]). 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 B which presents its most important features. The full code can be found on github[1].

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 (P-E. Dagand and C. McBride [4]) 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 N. Ghani et al ([12]).

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)[10] such as inductive types (W–types), inductive families *etc.* This rule has been

[1] `https://github.com/Lapin0t/induction-recursion`

inspired to P. Dybjer and A. Setzer ([5, 6]) 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 [7] 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 (B. Pierce and D. Turner [14]) 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 C.

Induction-recursion is arguably the most powerful set former (currently known) for ITT. A. Setzer ([15]) 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 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 (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`[2].

```
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)

_⇒_ : 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
```

2.3 A Universe of Strictly Positive Functors

P. Dybjer and A. Setzer have first presented codes for (indexed) inductive-recursive definitions by constructing a universe of functors. However, as conjectured by [12], this universe lacks closure under composition, *eg* if given the codes of two functors, we do not know how to construct a code for the composition of the functors. We will thus use an alternative universe construction devised by C. McBride ([12]) which we call *Irish* induction–recursion[3].

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[4] and a recursive predicate `info : poly γ X → Set`

[2]See section **??** 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.

[3]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.

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

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.
- 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
```

The functors FΣ and FΠ are functors that construct the dependent sum and dependent product of families, allowing us to construct families and arrows on them component by component. We will use them a couple more times in the same kind of structural recursion on poly.

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 do not 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
```

The post-composition functor we used is 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.

```
U₁ : Set
U₁ = μ-c ΠℕC *

`ℕ₁ : U₁
`ℕ₁ = ⟨ lift `ℕ , lift * ⟩

el₁ : U₁ ⟶ Set
el₁ = μ-d ΠℕC *

`Π₁ : (A : U₁) (B : el₁ A ⟶ U₁) ⟶ U₁
`Π₁ A B = ⟨ lift `Π , lift A , lift ∘ B ⟩
```

2.4.2 Catamorphism and Paramorphism

We previously said that this least–fixed point has in category theory the semantic of an initial algebra. Let us 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
```

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 $\text{fold}\ \rho$ is in fact a generic fold or with fancier wording an elimination rule, precisely the catamorphism (also called recursor). An elimination rule 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. Lets stop and write down the factorial function.

```
foldℕ : (f : X → X) (x : X) → ℕ → X
foldℕ f x zero = x
foldℕ f x (suc n) = f $ foldℕ f x n

_+ℕ_ : ℕ → ℕ → ℕ
_+ℕ_ = foldℕ suc

_*ℕ_ : ℕ → ℕ → ℕ
m *ℕ n = foldℕ (_+ℕ_ m) zero n

fact : ℕ → ℕ
fact zero = suc zero
fact (suc n) = suc n *ℕ fact n
```

One should be convinced that fact cannot be expressed as $\text{fold}\mathbb{N}\ f\ x$. Indeed for the $\text{suc}\ n$ case, besides the recursive call $\text{fact}\ n$, we need the unchanged data $\text{suc}\ n$. To solve this we introduce *paramorphisms* (the equivalent notion of primitive recursion in category theory). The specification is not an algebra $[\![\ \rho\ ]\!]\ X \Rightarrow X$ but an arrow $[\![\ \rho\ ]\!]\ (\mu\ \rho \times X) \Rightarrow X$, the domain of which is exactely a node where to every subnode we have added the recursive computation (but also left them in place). Note that there is no added power—only expressivity—since we can construct a fold with algebra $[\![\ \rho\ ]\!]\ (\mu\ \rho \times X) \Rightarrow \mu\ \rho \times X$ and drop the second component of the output. Every arrow $\mu\ \rho \Rightarrow X$ can be expressed as $\text{para}\ \varphi$ for some arrow φ (L. Merteens, [11]), as such it is the most expressive (non-dependent) eliminator. This expressivity of paramorphisms will be crucial in a later proof on ornaments.

```
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 $P : (i : \text{Code}\ X)\ (x : \text{Code}\ (\mu\ \rho\ i)) \to \text{Set}$, 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 us formalize that a bit. We define a predicate all stating that a property hold for all subnodes.

```
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 we 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
```

We 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 we 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 does not 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)
```

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

```
zip : 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[5]. 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
```

[5] 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.

```
    decode : (j : Code) → decode X (down j) → Set β₁
  open PRef public
```

Let $X : \mathsf{ISet}\ \alpha_0\ \beta_0$ and $R : \mathsf{PRef}\ \alpha_1\ \beta_1\ X$. $\mathsf{Code}\ R$ represents the new input index, together with the striping function $\mathsf{down}\ R$ taking new input indexes to old ones. Additionaly we have to define a new output index $Y : \mathsf{Code}\ R \rightarrow \mathsf{Set}$ such that we can derive a stripping function $(j : \mathsf{Code}\ R) \rightarrow Y\ j \rightarrow X\ (\mathsf{down}\ j)$. Directly defining Y together with this second striping function would not have been practical[6]. Thus instead of the stripping function, we ask for its fibers (called its graph), given by $\mathsf{decode}\ R$. This reversal is the classical choice between families $(A : \mathsf{Set}) \times A \rightarrow X$ and powersets $X \rightarrow \mathsf{Set}$ 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.

3.3 A Universe of Ornaments

Step by step, following the construction of induction–recursion, we will start by describing ornaments of poly, the inductive part of the definition. For $R : \mathsf{PRef}\ \alpha_1\ \beta_1\ X$ and $\rho : \mathsf{poly}\ \gamma_0\ X$ we define a universe of decriptions $\mathsf{orn}_0\ \gamma\ R\ \rho : \mathsf{Set}\ \_$. Simultaneously we 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 $\mathsf{info}\!\downarrow\ : \mathsf{info}\ \lfloor\ o\ \rfloor_0 \rightarrow \mathsf{info}\ \rho$ 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
```

[6] Later we would have needed to define preimages which necessarily embed some notion of equality. As explained in 4.1 we want to avoid any mention of equality when formalizing the unrelated matters of data types.

```
⌊_⌋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. ι j ornaments poly of the form ι i where down R j ≡ i *ie* we replace inductive positions by a fancy index such that the stripping matches. σ A B has to use the interpretation $\lfloor\_\rfloor_0$ and info↓ to express how the family B depends on the info of A. κ and σ 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.

$\lfloor\_\rfloor_0$ 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 ∗}
```

We 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
```

3.4 Ornamental Algebra

Recalling the first remark we made on the relation between an ornamented data type and its 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 do not 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.

```
forget : (o : orn γ₁ R R ρ) {s} → π₀ < (μ ⌊ o ⌋) ⇒ (μ ρ ∘ down R)
forget = ?
```

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 ⟦ ⌊ o ⌋ ⟧ (F ∘ down R) ⇒ (⟦ ρ ⟧ F ∘ 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 ∘ down R is not a valid argument to ⟦ ⌊ o ⌋ ⟧.

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 erase : ⟦ ⌊ o ⌋ ⟧ (F ∘ down R) ⇒ (μ ρ ∘ 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 ∘ down R does not contain enough information. In a fancy σ 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.

Note that a finer approach would be not to resort to fully featured paramorphisms. Indeed, to simplify a node we do not 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 its 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 Σ ℕ vec. More precisely to for each xs : list we can naturally associate xs′ : vec (length xs). length is no stranger, it is a very simple fold, *eg* the underlying core is an algebra ⟦ list − c ⟧ ℕ ⟹ ℕ. A natural generalization follows in which for a given algebra ⟦ ρ ⟧ X ⟹ X we create an ornament indexing elements of μ ρ 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 we 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 $[\![\ \rho\ ]\!]_0$ F to (orn_0 ($\gamma_0 \sqcup \alpha_1$) (AlgR F) ρ , λ o → (y : info ⌊ o $\rfloor_0$) → 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 $[\![\ \rho\ ]\!]_0$ 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 we introduce the type of fibers for a function[7].

```
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.

```
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 do not directly delegate the implementation of node to $algorn_0$. Indeed we have to come up with an element x : Code ($[\![\ \rho\ ]\!]_0$ 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. 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 _
```

[7]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.

```
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 ⟩
```

We have now finished all our constructions. To familiarize themselves further with them, the reader might continue with the case study in appendix C, studying an example formalization of simply–typed λ→ calculus.

4 Discussion

4.1 Index-First Datatypes and a Principled Treatment of Equality

Intuitionistic Type Theory has long realised the unsufficient study of the equality type, streched between a convenient extensional equality and complicated computational interpretation. Already in 2007, T. Altenkirch and C. McBride presented Observation Type Theory ([2]) which suggests an alternative to inductive propositional equality, which can be related to the non–higher–order fragment of the newer Homotopy Type Theory[8] by V. Voevodsky ([16]).

The inductive definition of propositional equality is deceptive on several matters. First it pollutes the formalization of datatypes, a matter which has no reason not to be orthogonal to equality. More than that, because we had no compelling alternative, the fantastic index–first presentation of datatypes with pattern matching on the index has been left behind. Indeed index–first presentation religously follows the bidirectional philosophy, ensuring that there cannot be several converging information flows triggering local definitional equality between expressions. This rules out every equality–like definition (like our definition of fiber) whose use is to pattern match on the proof to locally unify terms.

Regarding pattern matching on the index, from a very practical point of view it is reassuring that most types encountered in formal developments are not equality–like. When we do make use of input indexes depending on constructor argument, most of the time these arguments are marked implicit and this is the symptom of a hidden pattern matching: the two information flows do not really collide since we delegate one of the sides to the implicits–solving machinery. It is thus explicit that the only information flow indeed comes from the index, confirming its qualification as an *input* index. Acknowledging that internally in the language construction would mean cheap eradication of a big source of inefficiency that has already been investigated (E. Brady et al, [3]).

[8] Amusingly *OTT* is *HoTT* without the *H*.

Homotopy Type Theory seems to be where most of the research on equality is currently at, already with several experimental implementation namely one for Agda. Because of this promising ongoing research, now seems the good time to build tools that will enable the datatype theory to smoothly adapt to any new development of equality.

4.2 Further Work

I hit the surprising obstactle of forget not being a catamorphism quite late in the internship and as such, the study on paramorphisms is incomplete. The question of non–dependent elimination rules be further investigated.

In the same veine, the story for algebraic ornaments is missing a finishing touch. Given that we have formalized paramorphisms, there is a natural generalization from algebraic ornaments to paramorphic ornaments, possibly deriving the injection function for a wider array of ornaments. Additionally, it is unsatisfactory that reornaments are not yet able to make use of pattern–matching on the index to drive away more equality proofs (by eliminating contradictory information sources). Indeed in a reornament, we know the code of the index (which is the first in the sequence of the 3) and the erase algebra gives us the raw expression of the fold.

When we start to have first–class description of datatypes, a new world is open for exploration. G. Allais et al ([1]) have characterised the datatypes behaving like simply–typed syntaxes with binders. We might ask how it fits with this development. What is the best way for a language to expose primitive for syntax reflection, tying together the internal description of datatypes with their native counter–part?

Induction–recursion has recently been generalized even further than indexation by N. Ghani et al [8] in the form of induction–recursion for arbitrary fibers. Fibers are a category theory notion giving a general account of indexation. Indexed induction–recursion arises as a special case, but also setoid induction–recursion or category with families[9] induction–recursion (allowing one to define a universe equiped with a notion of substitution). This seems like an interesting step forward since by allowing one to explicitely state which *more specific than the most generic* notion of indexation we want, it degenerates gracefully (even to basic inductive types) with no need for the trivial indexation trick that we have used.

The last attack surface I can suggest is to work to achieving perhaps a more principled set of operations for the universe of ornaments as we have seen that they do not always mesh up very well and leave some trivial artifacts hanging when they are interpreted. A related question but which should not be taken as a reliable solution for the previous issue is the reorganization of datastructures, otherwise said the optimization of descriptions. Although this last project can probably only be effectively implemented in a language typechecker or depends on good reflection primitives.

A Bibliography

References

[1] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. 2:1–30, 07 2018.

[2] Thorsten Altenkirch and Conor McBride. Towards observational type theory. 01 2006.

[9] A model of type theories introduced by A. Setzer.

[3] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs, International Workshop, TYPES 2003*, Lecture Notes in Computer Science, pages 115–129, 04 2003.

[4] Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *CoRR*, abs/1201.4801, 2012.

[5] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *4th International Conference on Typed Lambda Calculi and Applications*, volume 1581, pages 129–146, 04 1999.

[6] Peter Dybjer and Anton Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1):1–47, 2003.

[7] Peter Dybjer and Anton Setzer. Indexed induction–recursion. *The Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.

[8] Neil Ghani, Lorenzo Malatesta, Fredrik Nordvall Forsberg, and Anton Setzer. Fibred data types. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 243–252, Washington, DC, USA, 2013. IEEE Computer Society.

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

[10] Per Martin-Löef and Giovanni Sambin. *Intuitionistic Type Theory*. Napoli : Bibliopolis, 1984.

[11] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[12] 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. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[13] 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.

[14] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[15] Anton Setzer. A model for a type theory with mahlo universe. 02 2001.

[16] V. Voevodsky. The equivalence axiom and univalent models of type theory. (Talk at CMU on February 4, 2010). *ArXiv e-prints*, 2014.

B Introduction to Agda

Good introductory material is available online[10]. We present here a speed–run through it.

[10] `http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf` (U. Norell and J. Chapman)

B.1 Syntax and concepts

Data types are introduced using the **data** keyword. Types are written in blue and constructors in red.

```
data bool : Set where
  true : bool
  false : bool
```

Set is the type of small types. There is a hierarchy of types Set : Set_1, Set_1 : Set_2 and so one. More on that later.

Total functions can be defined by pattern matching in a similar way to haskell by specifying several independent clauses. We write them in in green.

```
not : bool → bool
not true = false
not false = true

_&&_ : bool → bool → bool
true  && true  = true
true  && false = false
false && true  = false
false && false = false

if_then_else_ : {A : Set} → bool → A → A → A
if true  then a else b = a
if false then a else b = b
```

As we can see above, Agda has a powerful way of specifying mixfix operators, where arguments might be placed in order in place of underscores in the identifier. In other words, x && y is a shorthand for _&&_ x y. In fact almost every unicode character is valid in an identifier (apart from parenthesis, braces, dots, semicolons and at). The downside is that is a valid identifier and as such tokens must be separated by spaces.

Function expressions are introduced by the λ keyword: λ x → x. They can take several (curried) argument λ x y → y and can perform pattern matching when enclosed in braces: λ {true → false; false → true}.

Recursion, self or mutual does not have to be declared, the only requirement is scoping: an implementation has to follow (anywhere after) any declaration and for every identifier used, its declaration must preceed.

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

even : nat → bool
odd : nat → bool

even zero = true
even (suc n) = odd n

odd zero = false
odd (suc n) = even n
```

The dependent function type is written (x : A) → B where B may mention x.

```
id : {A : Set} → A → A
id x = x
```

As shown, implicit argument are marked by curly braces, we are not required to pass them when calling or defining the function and they will be solved by unification (not search). The ∀ symbol is a helper when we want to make the range of an argument implicit: we could have written id : ∀ {A} → A → A. Note that this also works with explicit arguments like id′ : ∀ A → A → A. We may drop arrows for dependent type: (X : Set) (Y : Set) → X → Y is a shorthand for (X : Set) → (Y : Set) → X → Y. We can resort to unification on explicit arguments by using an underscore in place of the argument, *eg* id′ _ x.

Records are introduced by the **record** and **field** keywords. We write projectors in orange.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    π₀ : A
    π₁ : B π₀
open Σ
```

The last line brings into scope the projectors π_0 : ∀ {A B} → Σ A B → A and π_1 : ∀ {A B} (p : Σ A B) → B (π_0 p). Before that we would have referred to them as Σ.π_0 and Σ.π_1. To construct an element there are 3 methods: using the defined constructor x , y, using generic record notation **record** {π_0 = x; π_1 = y} or by using copatterns (the preferred method, especially for the functions returning records):

```
p : Σ A B
π₀ p = foo
π₁ p = bar
```

B.2 Universe Levels

??

As explained previously, Agda has a tower of universes. The first ones have names like Set_2 but we can access any one by using levels (which are natural numbers where the constructors are axioms to disable pattern matching). The zero level is lzero and the sucessor is lsuc. We also have access to a max function _⊔_ : Level → Level → Level. We can write *level–polymorphic* functions.

```
id′ : {α : Level} {X : Set α} → X → X
id′ x = x
```

The tower of universes is not cumulative, if X : Set α, then we X : Set (lsuc α) is not true. This is particularly painful as it adds a lot of noise: to embed a small set into a higher one we have to resort to a record (or a datatype) as they can be given any level which is high enough.

```
record Lift (β : Level) (A : Set α) : Set (α ⊔ β) where
  constructor lift
  field lower : A
```

In the report i have hidden most prenex implicit arguments from function (using a mix of an existing feature resembling Coq's *Variable* and pure typographic hacks) as these are mostly related to level polymorphism bookkeeping. You should try to mentally remove every occurence of Lift, lift, lower and of level variables (to which I reserved the first 4 greek letters). *Ie* instead of ∀ {α β} → Set α → Set β → Set (α ⊔ β) I might write Set α → Set β → Set _.

B.3 Prelude

We will briefly introduce the most important utility definitions we will use throughout the report.

We already have seen the $\Sigma\ A\ B$ type with projectors π_0 and π_1. Its non–dependent counterpart is $\_\times\_ : \mathsf{Set}\ \alpha \rightarrow \mathsf{Set}\ \beta \rightarrow \mathsf{Set}\ \_$.

Level polymorphic empty and unit types:

```
data ⊥ {α} : Set α where
record ⊤ {α} : Set α where constructor *
```

Dependent function composition is written $g \circ f$ and dependent application is $f\ \$\ x$. I use this last definition a lot to escape a parenthesis hell.

Heterogeneous inductive equality is defined by:

```
data _≡_ (x : A) : B → Set α where
  refl : x ≡ x
```

We will use the usual lemmas $\mathsf{subst} : (P : A \rightarrow \mathsf{Set}\ \beta) \rightarrow x \equiv y \rightarrow P\ x \rightarrow P\ y$, $\mathsf{cong} : (f : (x : A) \rightarrow B\ x) \rightarrow x \equiv y \rightarrow f\ x \equiv f\ y$, $\mathsf{trans} : x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z$ and $\mathsf{sym} : x \equiv y \rightarrow y \equiv x$. Also their two argument version $\mathsf{subst}_2 : (P : (a : A) \rightarrow B\ a \rightarrow \mathsf{Set}\ \gamma) \rightarrow x_0 \equiv x_1 \rightarrow y_0 \equiv y_1 \rightarrow P\ x_0\ y_0 \rightarrow P\ x_1\ y_1$, $\mathsf{cong}_2 : \ldots$ and $\mathsf{cong}\text{–}\Sigma : \pi_0\ p \equiv \pi_0\ q \rightarrow \pi_1\ p \equiv \pi_1\ q \rightarrow p \equiv q$. We also make use of a postulated function extensionality:

```
postulate
  funext : {f : (x : A) → B₀ x} {g : (x : A) → B₁ x} → ((x : A) → f x ≡ g x) → f ≡ g
```

C Case Study: Bidirectional Simply-Typed Lambda Calculus

As an application of the theories that we constructed, we will present in this section a formalization of the bidirectional simply-typed $\lambda\!\rightarrow$ calculus. This will also provide a nice spot to take some time to motivate and explain bidirectional typing.

C.1 Bidirectional Typing

Bidirectional typing has been devised by B. Pierce and D. Turner ([14]) as a particular school of formalizing typing rules. Bidirectional typing has been particularly successful in taking over formalization but most importantly implementation of typecheckers for complex languages like dependent or substructural theories such as Agda itself or Idris. A motivation is the shortcoming of the Hindley–Milner algorithm for type inferance: in these theories a most generic type is usually not computable or may not even exist, yet we would like to avoid the necessity of annotating every single expression. Thus it arises with the need for a finer understanding of where type annotations are definitely not need and where they are, in the absence of an inferance engine.

Bidirectional typing emphasises the flow of information. One way to view a typechecker is as a server, responding to judgment queries either directly by a final answer or by a query itself, some sort of challenge. For example to the query "Does this variable x check to type T in context Γ?" a typechecker might offer responses such as "Yes, because $\mathsf{lookup}\ \Gamma\ x \equiv T$.", "Give me a proof that U is a type, $U <: T$ and $x : U$.". In these dialogs, we refer to input judgments as judgments implied by the hypothesis that the query is well-formed. A client might better be convinced that T is a valid type when asking if $x : T$ holds because the server will assume it. On the other hand, if the query is “What type has x?” then if given the answer T, the client can rightly assume that T is a valid type.

Precising things a bit we introduce not one but two typing judgement, with the information flow from left to right. $\Gamma \vdash x \in T$ represents the query “What is the type T that x has?” and $\Gamma \vdash T \ni x$ represents “Does x have type T”. The first mode of operation is called *synthesis*, with a $\top$ as input and a type as output and the second is called *checking*, with a type as input and $\top$ as output.

C.2 Native Agda

Before formalizing it with our encoding, we start of by giving the construction as we would normally in Agda. Let us start off by some tools. First natural numbers and finite sets.

```
data ℕ : Set where
  se : ℕ
  su : ℕ → ℕ

data fin : ℕ → Set where
  se : ∀ {n} → fin (su n)
  su : ∀ {n} → fin n → fin (su n)
```

Then contexts, also known as snoc–lists, together with a length, indexation and lookup.

```
data bwd (X : Set) : Set where
  ε : bwd X
  _,,_ : bwd X → X → bwd X

length : ∀ {X} → bwd X → ℕ
length ε = se
length (Γ ,, _) = su (length Γ)

idx : ∀ {X} (Γ : bwd X) → Set
idx Γ = fin (length Γ)

get : ∀ {X} (Γ : bwd X) → idx Γ → X
get (Γ ,, x) se = x
get (Γ ,, x) (su n) = get Γ n
```

The first judgements are type and env, giving the sets of types and valid contexts.

```
data type : Set where
  `base : type
  _`⇒_ : type → type → type
```

Know we can give the typing judgements. We will represent it by an indexed inductive–recursive type with as input index a context, a direction (synthesis or checking) and the associated input (type or $\top$, depending on the direction) and as output index the associated output (again type or $\top$).

```
data dir : Set where chk syn : dir

IN : dir → Set
IN chk = type
```

```
IN syn  =  ⊤

OUT : dir → Set
OUT chk  =  ⊤
OUT syn  =  type

data tlam₀ (Γ : env) : (d : dir) (i : IN d) → Set
out₀ : ∀ {Γ d i} → tlam₀ Γ d i → OUT d

aux : env → type → Set
aux Γ `base  =  ⊥ {lzero}
aux Γ (r `⇒ s)  =  tlam₀ Γ chk r

data tlam₀ Γ where
  lam : ∀ {r s} → tlam₀ (Γ ,, r) chk s            → tlam₀ Γ chk (r `⇒ s)
  vrf : ∀ {r} (e : tlam₀ Γ syn ∗) → out₀ e ≡ r  → tlam₀ Γ chk r
  var : idx Γ                                  → tlam₀ Γ syn ∗
  app : (f : tlam₀ Γ syn ∗) (x : aux Γ (out₀ f)) → tlam₀ Γ syn ∗
  ann : ∀ {r} → tlam₀ Γ chk r                  → tlam₀ Γ syn ∗

out₀ {Γ} {chk} {i} _           =  ∗
out₀ {Γ} {syn} {∗} (var i)     =  get Γ i
out₀ {Γ} {syn} {∗} (app f x)   with out₀ f | x
...                            | `base    | ()
...                            | r `⇒ s   | _  =  s
out₀ {Γ} {syn} {∗} (ann {r} _)  =  r
```

Let us make sense from this mess! Looking at the constructors, we have the usual lam, var and app. The constructor lam is in checking mode (it builds up larger types using small parts of given information) and the two destructors var and app (var can be interpreted as a destructor for the binding, app for the function themselves) are in synthesis mode as they take big arguments containing lots of information and represent smaller terms constrained by them.

There is a little trick in the type of app, indeed it is key to have the function argument f in synthesis mode, yet we want to *panic* when f does not synthetise a function type. For that we simply build a little helper that will match on the type and demand an element of the empty type when the type is `base. This way we are sure that no such element will be constructible.

The output function is trivial in the checking mode and shouldn't be challenging in the synthesis mode. We crucially make use of Agda's **with**–*abstraction*, a feature ressembling a case expression performed left of the clause equation (which do not exist natively in Agda).

C.3 Well–Scoped Terms

We do not want to directly jump to encoding this syntax of λ→ calculus because the funny part is that we will express it as an ornament on well–scoped syntax. Well–scoped syntax is expressed as an IIR definition with natural numbers as input index (the number of free variables) and a trivial output index.

```
ulam-ix : ISet lzero lzero
Code ulam-ix  =  ℕ
decode ulam-ix  =  λ _ → ⊤

data ulam-tag : Set where `var `app `lam `wrap : ulam-tag
ulam-c : IIR lzero ulam-ix ulam-ix
```

```
node ulam-c n  =  σ (κ ulam-tag) (λ {
   (lift `var)  ⟶  κ (fin n);
   (lift `app)  ⟶  σ (ι n) (λ _  ⟶  ι n);
   (lift `lam)  ⟶  ι (su n);
   (lift `wrap)  ⟶  ι n})
emit ulam-c n _  =  *

ulam  :  ℕ  ⟶  Set
ulam n  =  μ-c ulam-c n
```

There is one surprise: the `wrap constructor, that—as its name hints—does nothing really interesting, just adding a constructor layer for the sake of it. It is only here as an artifact of my definition of the universe of ornaments but I am not sure it could have been avoided. For now we can ignore it, the reason will appear in the following section.

C.4 Well–Typed Terms

First we give the reindexation and the constructor tags, the new indexes being as we have seen for tlam_0 and the stripping function being the length of the context.

```
tlam-ix  :  PRef lzero lzero ulam-ix
Code tlam-ix  =  env × Σ dir ℕ
down tlam-ix (Γ , _)  =  length Γ
decode tlam-ix (_ , d , _) _  =  OUT d

data syn-tag  :  Set where `var `app `ann  :  syn-tag
data chk-tag  :  Set where `lam `vrf  :  chk-tag
```

First let us look at the inductive part of the encoding.

```
ulam-c  :  orn lzero tlam-ix tlam-ix ulam-c
node ulam-c (Γ , chk , `base)  =
   σ (del-κ `wrap)
     λ _  ⟶  add₁ (ι (Γ , syn , *)) λ {(lift (_ , t))  ⟶  κ (t ≡ `base)}
node ulam-c (Γ , chk , r `⇒ s)  =  add₀ (κ chk-tag) λ {
   (lift `lam)  ⟶  σ (del-κ `lam) λ _  ⟶  ι (Γ , , r , chk , s);
   (lift `vrf)  ⟶  σ (del-κ `wrap) λ _  ⟶  add₁ (ι (Γ , syn , *)) λ {(lift (_ , t))  ⟶  κ (t ≡ r `⇒ s)}}
node ulam-c (Γ , syn , _)  =  add₀ (κ syn-tag) λ {
   (lift `var)  ⟶  σ (del-κ `var) λ _  ⟶  κ;
   (lift `app)  ⟶  σ (del-κ `app) λ _  ⟶  σ (ι (Γ , syn , *)) (λ {
     (lift (_ , `base))  ⟶  add₀ (κ ⊥) λ ();
     (lift (_ , (r `⇒ s)))  ⟶  ι (Γ , chk , r)});
   (lift `ann)  ⟶  σ (del-κ `wrap) λ _  ⟶  add₀ (κ type) (λ {(lift r)  ⟶  ι (Γ , chk , r)})}
```

The first comment is probably that this is a bit clumsy. We could've written special syntax rules to ease the programming with the encoding and the choice of operation for ornaments is not set in stone, it may be later changed to another combination.

We can note that we pattern match on the index, *eg* before giving the shape of the datatype (in this case the ornament). This is the full power of index–first datatypes unleashed, as such we have constructors that do not have any of the implicit quantification like in native Agda.

A pattern we notice is add_0 (κ...) λ {(lift...) ⟶ σ (del-κ...)...}. The high–level operation going on here is the replacement of some constant by another one (given a stripping function which is implicit here). We might want to add special syntax for that.

Now it is clear what `wrap stood for: the ornament introduces new constructors `vrf and `ann that do not exist in the original datatype. Without `wrap we wouldn't know what constructor to choose from in the old datatype. Note that this is an artifact in the sense that it might be avoidable. Indeed these added constructors do not really change the shape from untyped λ→ (without wrap), as they just add a *transparent* layer that we could very much erase systematically. It thus simply a matter of getting the axiom right and adding it as a constructor to orn_0.

Finishing with the unsurprising recursive part and the fixed–point:

```
emit ulam-c (Γ , chk , _) _ = *
emit ulam-c (Γ , syn , *) (lift `var , _ , lift (lift i)) = get Γ i
emit ulam-c (Γ , syn , *) (lift `app , _ , lift (_ , `base) , lift () , _)
emit ulam-c (Γ , syn , *) (lift `app , _ , lift (_ , (r `⇒ s)) , _) = s
emit ulam-c (Γ , syn , *) (lift `ann , _ , lift r , _) = r

ulam : (Γ : env) (d : dir) (i : IN d) → Set
ulam Γ d i = μ-c ⌊ ulam-c ⌋ (Γ , d , i)

out : ∀ {Γ d i} → ulam Γ d i → OUT d
out x = π₁ $ μ-d ⌊ ulam-c ⌋ _ x
```

We are now done! In the end the encoding has gone well but it stressed the need for syntactic sugar and it raised the issue of wrapper–like constructors that we should be allowed to add when ornamenting.