Running in circles in Agda

I'll have to grab a UU-template at some point

Samuel Klumpers
6057314

March 3, 2023

Contents

This document is generated from a literate agda file!

Abstract

The preliminary goal of this thesis is to introduce, among others, the concepts of the structure identity principle, numerical representations, and ornamentations, which are then combined to simplify the presentation and verification of finger trees, as a demonstration of the generalizability and improved compactness and security of the resulting code.

1 Introduction

Most of the time when we are Agda-ing [Tea23] we are trying to un-Haskell ourselves, e.g., not take the head of an empty list. In this example, we can make `head` total by switching to length-indexed lists: vectors. We have now effectively doubled the size of our code base, since functions like `_++_` which we had for lists, will also have to be reimplemented for vectors.

To make things worse; often, after coping with the overloaded names resulting from Agda-ing by shoving them into a different namespace, we also find out that lists nor vectors are efficient containers to begin with. Maybe binary trees are better. We now need four times the number of definitions to keep everything working, and, if we start proving things, we will also have to prove everything fourfold. (Not to mention that reasoning about trees is probably going to be harder than reasoning about lists). This inefficiency has sparked (my) interest in ways to deal with the situation.

Following [DM14] and [KG16], we can describe the relation between list and vector using the mechanism of ornamentation. This leads them to define the concept of patches, which can aid us when defining `_++_` for the second time by forcing the new version to be coherent. In fact, the algebraic nature of ornaments can even get us the definition of the vector type for free, if we started by defining lists relative to natural numbers [McB14]. Such constructions rely heavily on descriptions of datastructures and often come with limitations in their expressiveness. These descriptions in turn impose additional ballast on the programmer, leading us to investigate reflection like in [EC22] as a means to bring datatypes and descriptions closer when possible.

From a different direction, [HS22] gives methods by which we can show two implementations of some structure to be equivalent. With this, we can simply transport all proofs about `_++_` we have for lists over to the implementation for trees, provided that we show them to be equivalent as appendable containers. This process can also be automated by some heavy generics, but instead, we resort to cubical; which hosts a range of research like [Ang+20] tailored to the problem describing equivalences of structures.

We can liken the situation to movement on a plane, where ornamentation moves us vertically by modifying constructors or indices, and structured equivalences move us horizontally to and from equivalent but more equivalent implementations. In this paper, we will investigate a variety of means of moving around structures and proofs, and ways to make this more efficient or less intrusive.

Currently, all sections mainly reintroduce or reformulate existing research, with some spots of new ideas and original examples here and there. In section 2, we will look at how proofs on unary naturals can be moved to binary naturals. Then in section 3 we recall how numeral systems in particular induce container types, which we attempt to reformulate in the language of ornaments in section 4.

2 How Cubical Agda helps our binary numbers (ready)

Let us quickly review the small set of features in Cubical Agda that we will be using extensively throughout this article.[1] We note that there are some downsides to cubical, such as that

```
{-# OPTIONS --cubical #-}
```

also implies the negation of axiom K, which in turn complicates both some termination checking and some universe levels. And, more obviously, we get saddled with the proof obligation that our types are sets should we use certain constructions.

Not sure if it would be helpful to have a more extensive introduction covering all features used.

Of course, this downside is more than offset by the benefits of changing our primitive notion of equality, which among other things, lets us access univalence, which drastically cut down the investment required to both show more complex structures to be equivalent (at least when compared to non-cubical). Here, equality arises not (directly) from the indexed inductive definition we are used to, but rather from the presence of the interval type I. This type represents a set of two points i0 and i1, which are considered "identified" in the sense that they are connected by a path. To define a function out of this type, we also have to define the function on all the intermediate points, which is why we call such a function a "path". Terms of other types are then considered identified when there is a path between them.

To be precise, the mapFold in [KG16] gets painted red.

As an added benefit, this different perspective gives intuitive interpretations to some proofs of equality, like

```
sym : x ≡ y → y ≡ x
sym p i = p (~ i)
```

where ~_ is the interval reversal, swapping i0 and i1, so that sym simply reverses the given path.

Furthermore, because we can now interpret paths in records and function differently, we get a host of "extensionality" for free. For example, a path in $A \to B$ is indeed a function which takes each i in I to a function $A \to B$. Using this, function extensionality becomes tautological

```
funExt : (∀ x → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

Finally, the Glue type tells us that equivalent types fit together in a new type, in a way that guarantees univalence

[1] [VMA19] gives a comprehensive introduction to cubical agda.

```
ua : ∀ {A B : Type ℓ} → A ≃ B → A ≡ B
```
For our purposes, we can interpret univalence as "equivalent types are identified", and, we can treat equivalences as the "HoTT-compatible" generalization of bijections. In particular, type isomorphisms like $1 \to A \simeq A$ actually become paths $1 \to A \equiv A$, such that we can transport proofs along them. We will demonstrate this by a slightly more practical example.

2.1 Binary numbers

Let us motivate the cubical method by showing the equivalence of the "Peano" naturals and the "Leibniz" naturals. Recall that the Peano naturals are defined as

```
data ℕ : Type where
  zero : ℕ
  suc : ℕ → ℕ
```
This definition enjoys a simple induction principle and has many proofs of its properties in standard libraries. However, it is too slow to be of practical use: most arithmetic operations defined on ℕ have time complexity in the order of the value of the result.

Of course, the alternative are the more performant binary numbers: the time complexities for binary numbers are usually logarithmic in the resultant values, but these are typically less well-covered in terms of proofs. This does not have to be a problem, because the ℕ naturals and the binary numbers should be equivalent after all!

Let us make this formal. We define the Leibniz naturals as follows:

is this too much code and too little explanation at once?

```
data Leibniz : Set where
  0b : Leibniz
  _1b : Leibniz → Leibniz
  _2b : Leibniz → Leibniz
```
Here, the 0b constructor encodes 0, while the _1b and _2b constructors respectively add a 1 and a 2 bit, under the usual interpretation of binary numbers:

```
toℕ : Leibniz → ℕ
toℕ 0b = 0
toℕ (n 1b) = 1 N.+ 2 N.· toℕ n
toℕ (n 2b) = 2 N.+ 2 N.· toℕ n
```
Let us construct the equivalence from ℕ to Leibniz. First, we can also interpret a number in ℕ as a binary number by repeating the successor operation on binary numbers:

```
bsuc : Leibniz → Leibniz
bsuc 0b = 0b 1b
bsuc (n 1b) = n 2b
bsuc (n 2b) = (bsuc n) 1b

fromℕ : ℕ → Leibniz
fromℕ ℕ.zero = 0b
```

```
    fromℕ (ℕ.suc n) = bsuc (fromℕ n)
```
To show that the operations are inverses, we observe that the interpretation respects successors
```
    toℕ-suc : ∀ x → toℕ (bsuc x) ≡ ℕ.suc (toℕ x)
    toℕ-suc 0b = refl
    toℕ-suc (x 1b) = refl
    toℕ-suc (x 2b) = cong
      (λ k → (1 N.+ 2 N.· k))
      (toℕ-suc x) · cong ℕ.suc (NP.·-suc 2 (toℕ x))
```
and that the inverse respects even and odd numbers
```
    fromℕ-1+2· : ∀ x → fromℕ (1 N.+ 2 N.· x) ≡ (fromℕ x) 1b
    fromℕ-1+2· ℕ.zero = refl
    fromℕ-1+2· (ℕ.suc x) = cong
      (bsuc ∘ bsuc)
      (cong fromℕ (NP.+-suc x (x N.+ ℕ.zero)) · fromℕ-1+2· x)

    fromℕ-2+2· : ∀ x → fromℕ (2 N.+ 2 N.· x) ≡ (fromℕ x) 2b
    fromℕ-2+2· x = cong bsuc (fromℕ-1+2· x)
```
The wanted statement follows
```
    ℕ↔L : Iso ℕ Leibniz
    ℕ↔L = iso fromℕ toℕ sec ret
      where
      sec : section fromℕ toℕ
      sec 0b = refl
      sec (n 1b) = fromℕ-1+2· (toℕ n) · cong _1b (sec n)
      sec (n 2b) = fromℕ-2+2· (toℕ n) · cong _2b (sec n)

      ret : retract fromℕ toℕ
      ret ℕ.zero = refl
      ret (ℕ.suc n) = toℕ-suc (fromℕ n) · cong ℕ.suc (ret n)
```
but since we now have a bijection, we also get an equivalence
```
    ℕ≃L : ℕ ≃ Leibniz
    ℕ≃L = isoToEquiv ℕ↔L
```
Finally, by univalence, we can identify ℕ and Leibniz naturals
```
    ℕ≡L : ℕ ≡ Leibniz
    ℕ≡L = ua ℕ≃L
```
Using the path ℕ≡L we can already prove some otherwise difficult properties, e.g.,
```
    isSetL : isSet Leibniz
    isSetL = subst isSet ℕ≡L N.isSetℕ
```
Let us define an operation on Leibniz and demonstrate how we can also transport proofs about operations from ℕ to Leibniz.

2.2 Use as definition: functions from specifications

As an example, we will define addition of binary numbers. We could take

```
BinOp : Type → Type
BinOp A = A → A → A

_+′_ : BinOp Leibniz
_+′_ = subst BinOp ℕ≡L N._+_
```

But this would be rather inefficient, incurring an $O(n+m)$ overhead when adding $n+m$, so we could better define addition directly. We would prefer to give a definition which makes use of the binary nature of Leibniz, while agreeing with the addition on ℕ.

Such a definition can be derived from the specification "agrees with _+_", so we implement the following syntax for giving definitions by equational reasoning, inspired by the "use-as-definition" notation from [HS22]:

```
Def : {X : Type a} → X → Type a
Def {X = X} x = Σ' X λ y → x ≡ y

defined-by : {X : Type a} {x : X} → Def x → X
defined-by = fst

by-definition : {X : Type a} {x : X} → (d : Def x) → x ≡ defined-by d
by-definition = snd
```

which infers the definition from the right endpoint of a path using an implicit pair type

```
record Σ' (A : Set a) (B : A → Set b) : Set (ℓ-max a b) where
  constructor _use-as-def
  field
    {fst} : A
    snd : B fst

open Σ'

infix 1 _use-as-def
```

As of now, I am unsure if this reduces the effort of implementing a coherent function, or whether it is more typically possible to give a smarter or shorter proof by just giving a definition and proving an easier property of it[2]

With this we can define addition on Leibniz and show it agrees with addition on ℕ in one motion

```
{-# TERMINATING #-}
plus-def : ∀ x y → Def (fromℕ (toℕ x N.+ toℕ y))
plus-def 0b y       = ℕ↔L .rightInv y use-as-def
plus-def (x 1b) 0b =
  bsuc (fromℕ (toℕ x N.+ (toℕ x N.+ ℕ.zero) N.+ ℕ.zero))
    ≡⟨ cong (bsuc ∘ fromℕ) (NP.+-zero (2 N.· toℕ x)) ⟩
  bsuc (fromℕ (toℕ x N.+ (toℕ x N.+ ℕ.zero)))
```

[2]I will put the alternative in the appendix for now

```
      ≡⟨ fromℕ-1+2· (toℕ x) ⟩
    fromℕ (toℕ x) 1b
      ≡⟨ cong _1b (ℕ↔L .rightInv x) ⟩
    x 1b ■ use-as-def
  plus-def (x 1b) (y 1b) =
    fromℕ ((1 N.+ 2 N.· toℕ x) N.+ (1 N.+ 2 N.· toℕ y))
      ≡⟨ cong fromℕ (Eq.eqToPath (eq (toℕ x) (toℕ y))) ⟩
    fromℕ (2 N.+ (2 N.· (toℕ x N.+ toℕ y)))
      ≡⟨ fromℕ-2+2· (toℕ x N.+ toℕ y) ⟩
    fromℕ (toℕ x N.+ toℕ y) 2b
      ≡⟨ cong _2b (by-definition (plus-def x y)) ⟩
    defined-by (plus-def x y) 2b ■ use-as-def
      where
      eq : ∀ x y
         → (1 N.+ 2 N.· x) N.+ (1 N.+ 2 N.· y) Eq.≡ 2 N.+ (2 N.· (x N.+ y))
      eq = NS.solve-∀
  -- similar clauses omitted

  plus : ∀ x y → Leibniz
  plus x y = defined-by (plus-def x y)

  plus-coherent : ∀ x y → fromℕ (x N.+ y) ≡ plus (fromℕ x) (fromℕ y)
  plus-coherent x y = cong fromℕ
    (cong₂ N._+_ (sym (ℕ↔L .leftInv x)) (sym (ℕ↔L .leftInv _))) ·
      by-definition (plus-def (fromℕ x) (fromℕ y))
```

2.3 Structure Identity Principle

We see that as a consequence (modulo some PathP lemmas), we get a path from (ℕ, N.+) to (Leibniz, plus). More generally, we can view a type X combined with a function $f : X \to X \to X$ as a kind of structure, which in fact coincides with a magma. We can see that paths between magmas correspond to paths between the underlying types X and paths over this between their operations f. This observation is further generalized by the Structure Identity Principle (SIP), formalized in [Ang+20]. Given a structure, which in our case is just a binary operation

Replace with BinOp

```
  MagmaStr : Type → Type
  MagmaStr A = A → A → A
```

this principle produces an appropriate definition "structured equivalence" ι. The ι is such that if structures X, Y are ι-equivalent, then they are identified. In this case, the ι asks us to provide plus-coherent, so we have just shown that the plus magma on Leibniz

```
  MagmaL : Magma
  fst MagmaL = Leibniz
  snd MagmaL = plus
```

and the _+_ magma on ℕ and are identical

```
Magmaℕ≃MagmaL : Magmaℕ ≡ MagmaL
Magmaℕ≃MagmaL = equivFun (MagmaΣPath _ _) proof
  where
  proof : Magmaℕ ≃[ MagmaEquivStr ] MagmaL
  fst proof = ℕ≃L
  snd proof = plus-coherent
```

As a consequence, associativity of plus for Leibnizfollows immediately from that of _+_ on ℕ:

```
plus-assoc : Associative _≡_ plus
plus-assoc = subst
  (λ A → Associative _≡_ (snd A))
  MagmaℕMagmaL
  ℕ-assoc
```

3 Types from specifications: Numerical representations (rough)

Perhaps the conclusion from the last section was not very thrilling, especially considering that ℕ is a candidate to be replaced by a more suitable unsigned integer type when compiling to Haskell anyway. More relevant to the average Haskell programmer are containers, and their associated laws.

As an example in the same vein as the last section, we could define a type of inefficient lists, and then define a type of more efficient trees. We can show the two to be equivalent again, so that if we show that lists trivially satisfy a set of laws, then trees will satisfy them as well. But even before that, let us reconsider the concept of containers, and inspect why trees are more efficient than lists to begin with.

Rather than defining inductively defining a container and then showing that it is represented by a lookup function, we can go the other way and define a type by insisting that it is equivalent to such a function. This approach, in particular the case in which one calculates a container with the same shape as a numeral system was dubbed numerical representations in [Oka98], and is formalized in [HS22]. Numerical representations form the starting point for defining more complex datastructures based off of simpler basic structures, so let us run through an example.

3.1 Vectors from Peano

We can compute the type of vectors starting from the Peano naturals[3]. For simplicity, we define them as a type computing function via the "use-as-definition" notation from before. Recall that we expect $VAn = Finn- > A$, so we should define $Finn$ first. In turn $Finn = \Sigma[m \in N]m < n$.

fix the in-line code

fix this code

[3]This is adapted (and fairly abridged) from [HS22]

SIP doesn't mesh very well with indexed stuff, does HSIP help?

We can implement some basic operations (e.g., lookup and tail) on the representable arrays, and show some of their properties. Again, we can transport these proofs to vectors.

do this

(This computation can of course be generalized to any arity zeroless numeral system; unfortunately beyond this set of base types, this "straightforward" computation from numeral system to container loses its efficacy. In a sense, the n-ary natural numbers are exactly the base types for which the required steps are convenient type equivalences like $(A + B) - > C = A - > C \times B - > C$?)

4 Relating types by structure: Ornamentation (unfinished)

4.1 Numerical representations as ornaments

We could vigorously recompute a bunch of datastructures from their numerical representation, but we note that these computations proceed with roughly the same pattern: each constructor of the numeral system gets assigned a value, and is then replaced by a constructor which holds elements and subnodes using this value as a "weight". But wait! The "modification of constructors" is already made formal by the concept of ornamentation!

Ornamentation, as exposed in [McB14] and [KG16], lets us formulate what it means for two types to have "similar" recursive structure. This is achieved by interpreting (indexed inductive) datatypes from descriptions, between which an ornament is seen as a certificate of similarity, describing which fields or indices need to be introduced or dropped. Furthermore, a one-sided ornament: an ornamental description, lets us describe new datatypes by recording the modifications to an existing description.

add a small listing for this?

This links back to the construction in the previous section, since Nat and Vec share the same recursive structure, so Vec can be formed by introducing indices and adding a field A at each node.

include this

We can already tell that attempting the same for trees and binary numbers fails: they have very different recursive structures! Still, the correct tree constructors relate to those of binary numbers via the size of the resultant tree. In fact, this relation is regular enough that we can "fold in" trees into a structure which *can* be described as an ornament on binary numbers.

i'm pretty sure okasaki already knew the fix

4.2 Folding in

Let us describe this procedure of folding a complex recursive structure into a simpler structure more generally, and relate this to the construction of binary heaps in [KG16].

go

5 Equivalence from initiality (where does this go?)

6 Is equivalence too strong (finger trees)

7 Discussion and future work (aka the union of my to-do list and the actual future work section)

8 Temporary

Todo list

References

[Ang+20] Carlo Angiuli et al. *Internalizing Representation Independence with Univalence.* 2020. DOI: 10.48550/ARXIV.2009.05547. URL: https://arxiv.org/abs/2009.05547.

[DM14] PIERRE-ÉVARISTE DAGAND and CONOR McBRIDE. "Transporting functions across ornaments". In: *Journal of Functional Programming* 24.2-3 (Apr. 2014), pp. 316–383. DOI: 10.1017/s0956796814000069. URL: https://doi.org/10.1017%2Fs0956796814000069.

[EC22] Lucas Escot and Jesper Cockx. "Practical Generic Programming over a Universe of Native Datatypes". In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547644. URL: https://doi.org/10.1145/3547644.

[HS22] Ralf Hinze and Wouter Swierstra. “Calculating Datastructures”. In: *Mathematics of Program Construction*. Ed. by Ekaterina Komendantskaya. Cham: Springer International Publishing, 2022, pp. 62–101. ISBN: 978-3-031-16912-0.

[KG16] HSIANG-SHANG KO and JEREMY GIBBONS. “Programming with ornaments”. In: *Journal of Functional Programming* 27 (2016), e2. DOI: `10.1017/S0956796816000307`.

[McB14] Conor McBride. “Ornamental Algebras, Algebraic Ornaments”. In: 2014.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. USA: Cambridge University Press, 1998. ISBN: 0521631246.

[Tea23] Agda Development Team. *Agda*. 2023. URL: `https://agda.readthedocs.io/en/v2.6.3/`.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types”. In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: `10.1145/3341691`. URL: `https://doi.org/10.1145/3341691`.