Running in circles in Agda

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

Samuel Klumpers
6057314

February 28, 2023

Contents

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 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.

Before we try to improve or generalize upon these approaches, let us clarify some parts of the enviroment we are working in, partially by going through some examples.

2 How Cubical Agda helps our binary numbers (almost done)

Let us quickly review the small set of features in Cubical Agda that we will be using extensively throughout this article.[1] At the surface, there are two significant differences. The downside, being that

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

also implies the negation of axiom K, which in turn sabotages both some termination checking and some universe levels. And, more obviously, we get saddled

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

with the proof obligation that our types are sets should we use certain constructions.

The upside is that we get a primitive notion of equality, and get access to univalence which both drastically cut down the investment required to both show more complex structures to be equivalent (at least when compared to non-cubical) and also use this equivalence meaningfully. 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 such a function represents a "path". Terms of other types are then considered identified when there is a path between them.

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 its path argument.

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. Using this, proving function extensionality becomes easy enough:

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

Finally, univalence tells us that when two types are equivalent, then they can also be identified. For our purposes, we can treat equivalences as the "HoTT-compatible" generalization of bijections. In particular, type isomorphisms like $1 \to A \simeq A$ actually become equalities $1 \to A \equiv A$, such that we can transport proofs along them. We will demonstrate this by a slightly more practical example.

put the ua thing here

2.1 Binary numbers

We will take a look the underlying shapes of lists and trees, namely, the (Peano) natural numbers and the (Leibniz) binary natural numbers. We define the Leibniz naturals as follows:

```
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:

```
⟦_⟧ : Leibniz → ℕ
⟦ 0b ⟧ = 0
```

```
⟦ n 1b ⟧ = 1 ℕ.+ 2 ℕ.· ⟦ n ⟧
⟦ n 2b ⟧ = 2 ℕ.+ 2 ℕ.· ⟦ n ⟧
```

Clearly ℕ and Leibniz "are the same", so let us summarize this proof. 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

```
⟦⟧-suc : ∀ x → ⟦ bsuc x ⟧ ≡ ℕ.suc ⟦ x ⟧
⟦⟧-suc 0b = refl
⟦⟧-suc (x 1b) = refl
⟦⟧-suc (x 2b) = cong (λ k → (1 ℕ.+ 2 ℕ.· k)) (⟦⟧-suc x) · cong ℕ.suc (NP.·-suc 2 ⟦ x ⟧)
```

and that the inverse respects even and odd numbers

line too long

```
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 ⟦_⟧ sec ret
  where
  sec : section fromℕ ⟦_⟧
  sec 0b = refl
  sec (n 1b) = fromℕ-1+2· ⟦ n ⟧ · cong _1b (sec n)
  sec (n 2b) = fromℕ-2+2· ⟦ n ⟧ · cong _2b (sec n)

  ret : retract fromℕ ⟦_⟧
  ret ℕ.zero = refl
  ret (ℕ.suc n) = ⟦⟧-suc (fromℕ n) · cong ℕ.suc (ret n)
```

but since we now have a bijection, we also get an equivalence

fix this symbol

```
ℕ≃L : ℕ ≃ Leibniz
ℕ≃L = isoToEquiv ℕ↔L
```

Finally, by univalence, we can identify the Peano and Leibniz naturals

```
ℕ≡L : ℕ ≡ Leibniz
ℕ≡L = ua ℕ≃L
```

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

2.2 Structure Identity Principle

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
```

It would be a shame if we defined binary numbers and identified them with the naturals and then proceeded to not use them. So, let us define addition. Clearly, a sensible implementation should agree with natural addition under the interpretations. 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.

Inspired by the "use-as-definition" notation from [HS22], we define the following syntax for giving definitions by equational reasoning

define it

with which we can define

define plus

We see that as a consequence, the pairs $(, N.+)$ and $(Leibniz, +)$ are equal. More generally, we can view a type X combined with a function $\_\cdot\_ : X \to X \to X$ as a kind of structure, which in fact coincides with a magma. Now we can see that two magmas are identical if their underlying types X are, and their operations $\_\cdot\_$ agree modulo their identification. This observation is further generalized by the Structure Identity Principle (SIP), formalized in [Ang+20]. This principle describes a procedure to derive for a structure the appropriate notion of "structured equivalence" ι. The ι is such that if structures X, Y are ι-equivalent, then they are identified.

In our case, we have just shown that the _+_ magmas on ℕ and Leibniz are equal, hence associativity of _+_ for Leibniz follows immediately from that on ℕ.

prove it

3 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 [purely functional], and is formalized in [calcdata]. Numerical representations form the starting point for defining more complex datastructures based off of simpler basic structures, so let us run through an example.

fix citations

3.1 Vectors from Peano

decide on consistently using Peano or N

We can compute the type of vectors starting from the Peano naturals [this is worked out in full detail in calcdata]. 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 citation

SIP doesn't mesh very well with indexed stuff, does HSIP help? We can some basic operations [lookup/tail] and show some properties. Again, we can transport these proofs to vectors.

fix the inline code

(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 $(A + B)- > C = A- > C \times B- > C$?)

fix this code

do this

4 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!

fix citations

Ornamentation, exposed in [algebraic ornaments; progorn], 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.

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.
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.

4.2 Folding in

fix citations

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 [progorn].

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.

[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.