Bialgebras for simple distributive laws – Definitions and proofs, formalized

Cass Alexandru

March 2023

1 Introduction

The original goal of this research internship was to formalize some of the proofs from “Sorting with bialgebras and distributive laws”, which uses bialgebras and distributive laws to analyze sorting algorithms from a category theory perspective. I was planning to use existing definitions from the `agda-categories`[3] library, but it quickly became apparent that:

a) I would need to refactor some existing library code;

b) what I began formalizing were the abstract categorical notions, independent of their application to sorting, which as such belonged in the library itself.

I ended up formalizing only definitions and proofs about the abstract notion of bialgebras for a simple distributive law, and so was able to contribute my entire formalization to the `agda-categories` library. The code I contributed can be found at the pertinent pull-request: “`μ-Bialgebras by cxandru · Pull Request #362 · agda/agda-categories`”.

During the course of my internship, I was made aware of [1] and [4] as more abstract categorical references for the theory of bialgebras & distributive laws. Those two papers present the the theory in the context of its applications to operational semantics, which is a great illustration of a single categorical notion having many varied applications.

2 Background, Definitions and Notation

2.1 Maths

2.1.1 Background

In theoretical computer science, syntax has long been modelled via algebras. An example of syntax is the grammar of regular expressions for an alphabet A, which has the signature $\Sigma = \{\emptyset : 0, \varepsilon : 0, \text{literal} : 0, \text{concat} : 2, | : 2, * : 1\}$. Regular expression terms and their interpretation as languages form algebras for the functor: $R(X) = 1 + 1 + A + (X \times X) + (X \times X) + X$. Universal coalgebra [7] is an approach of modelling many kinds of transition systems via coalgebras. An example of transition systems are deterministic finite automata for an alphabet A. They can be described as coalgebras for the functor $(-)^A \times 2$ (where the $- \times 2$ encodes whether a state is accepting and $(-)^A$ encodes the label with which a transition happens). The seminal work [8] by Turi and Plotkin used this way of modeling syntax and behavior to describe SOS (structural operational semantics) specifications as *distributive* laws of syntax over behavior functors. For more background on the history and development of these theoretical notions in the field of operational semantics, see [4].

In the field of programming language analysis and design, *algebraic data types* have been modeled as fixpoints for so-called shape/base functors [5]. The well-known cons-list for elements of type A e.g. can be modelled as the fixpoint of the functor $L(X) = 1 + A \times X$. In languages without termination requirements (languages, that is, with a complete partial order domain semantics) the least and greatest fixpoints coincide, which means that data types are carriers of both the initial algebra and the final coalgebra for their base functor. This way of modelling recursive data types allows for describing functions *into* such a type as coinductive extensions of coalgebras, and functions out of them as inductive extensions of algebras, a method which was popularized in [6]. In [2], sorting algorithms are built up of functions that destruct an input type to construct an output type, described by distributive laws of the input type's base functor over that of the output type.

2.1.2 Definitions and Notation

1. Given two endofunctors T and F on a category $\mathcal{C}$, we can define a (T, F)-Bialgebra to be an object of $\mathcal{C}$ equipped with the structure of a T-Algebra and an F-Coalgebra, i.e. a triple (A, a, c), where $a : TA \to A$ and $c : A \to FA$. μ [1] is a distributive law (of T over F): a natural transformation $\mu \colon TF \Rightarrow FT$. A (T, F, μ)-Bialgebra is a (T, F)-Bialgebra (A, a, c) such that $c \circ a = Fa \circ \mu_A \circ Tc$. In the following, we may refer to T as the "algebraic" functor and to F as the "coalgebraic" functor. Note: In the context of bialgebras for operational semantics, these functors are often referred to as the "syntax" and "behaviour" functor, respectively [4].

2. $\mathsf{Bialgs}(T, F, \mu)$ is the category of (T, F, μ)-Bialgebras. In addition to the explicit construction of $\mathsf{Bialgs}(T, F, \mu)$ as outlined above, there are two other ways to construct it:

[1] Note that in the literature the distributive law is usually called λ, however λ is a keyword for introducing function abstraction in Agda, therefore I opted to go for μ instead. Jurriaan Rot rightfully pointed out to us that μ is often used to refer to the multiplication operation of monads, and suggested I use δ. However, I had already written a considerable amount of code by that point. Additionally, "delta" is often associated with a unit of change in common parlance (from physics). Therefore, I stuck with μ.

   a) We can lift F via μ to the category of $\mathsf{Algs}(T)$ (see `Categories.Functor.Construction.LiftAlgebras`), giving us the functor: $\hat{F}\colon \mathsf{Algs}(T) \to \mathsf{Algs}(T)$, where $\hat{F}_0\colon TA \xrightarrow{a} A \mapsto TFA \xrightarrow{\mu_A} FTA \xrightarrow{Fa} FA$.

   b) Dually we can lift T via μ to the category of $\mathsf{Coalgs}(F)$ (see`Categories.Functor.Construction.LiftCoalgebras`), giving us the functor: $\hat{T}\colon \mathsf{Coalgs}(F) \to \mathsf{Coalgs}(F)$, where $\hat{T}_0\colon C \xrightarrow{c} FC \mapsto TC \xrightarrow{Tc} TFC \xrightarrow{\mu_C} FTC$.

   Then the categories of $\mathsf{Coalgs}(\hat{F})$ and $\mathsf{Algs}(\hat{T})$ are two other ways of constructing the category $\mathsf{Bialgs}(T, F, \mu)$.

3. Given an initial object $\langle \bot, \mathrm{in} \rangle$ in the category of $\mathsf{Algs}(F)$ and a target Algebra $\langle A, a \rangle$, we denote the unique F-Algebra-Morphism $(\!| a |\!)\colon \bot \to A$.
   Given a terminal object $\langle \top, \mathrm{out} \rangle$ in the category of $\mathsf{Coalgs}(F)$ and a source Coalgebra $\langle C, c \rangle$, we denote the unique F-Coalgebra-Morphism $[\![c]\!]\colon C \to \top$.
   These notations for the initial/terminal morphisms are taken from [6] and are called "banana brackets" and "lenses" respectively.

4. We may, as [2] call initial T-Algebras μT and terminal F-Coalgebras νF. These are the usual notations for the least/greatest fixpoints of functors. These constructions are specific to certain types of categories; we will be using them here however to refer to *any* initial/terminal (Co)Algebras, as initial/terminal objects always appear in contravariant positions in the propositions we treat.

5. We may refer to a (Co)Algebra $\langle A, a \rangle$ interchangeably by its carrier (A) or associated morphism (a). This is a notational ambiguity that is not granted us in the Agda code.

2.2 Agda

2.2.1 Background

Agda is a dependently typed programming language and proof assistant. Unlike other proof assistants, there is no tactic language, so one has to provide proof terms directly. Interactivity is provided via typed holes that one can *refine* in the interactive editor mode. As a programming language, it has an advanced *module system*, of which I made heavy use in my code. The reason I chose Agda and not, e.g. Coq, was that I hadn't used it before and had the ambition to learn it, for which this project seemed a perfect opportunity.

2.2.2 Notation

- Identifiers in Agda may contain any unicode characters, including previously operators. I.e. e.g. `Foo⇒Bar`, `foo∘bar` are valid identifier names and are not to be confused with the terms `Foo ⇒ Bar`, `foo ∘ bar`

- We may define mixfix operators by putting underscores in argument positions of an identifier name. So `_+_` defines an infix operator, and `⦇_⦈` a paired delimiter.

- we call the category Algs(F^{op}) `coF-Algebras` to distinguish it from Coalgs(F) (`F-Coalgebras`).
- `A,α` are the fields for the carrier and associated morphism of a (Co)Algebra.
- `f,commutes` are the fields for the $\mathcal{C}$-Morphism and commutativity condition of a (Co)Algebra-Morphism.
- `η` is the field for instantiating a natural transformation at an object.

3 Contributions

I formalized all the definitions (Section 2 1,2) and a number of proofs pertaining to bialgebras. Using [1] as reference, these include Definition 3.2.2, Theorem 3.2.3 (with lemmata 3.2.4 and 3.2.5). I also proved, using [4] as a reference, Proposition 12 and a version of Theorem 13 (which is the same as [2, p. 73]). I used the preexisting `Algebra`, `Coalgebra` and `F-Algebras` modules, but refactored the `(Co)Algebra` modules and defined `F-Coalgebras`, dualizing proofs from `F-Algebras`. I sucessfully contributed the entirety of my formalization to the `agda-categories` library. In the following I outline details of the formalization I deem noteworthy.

3.1 Refactoring `(Co)Algebra`

Originally, the module `Categories.Functor.Algebra` contained the following definition:

```
record F-Algebra (F : Endofunctor C) : Set (o ⊔ ℓ) where
    open Category C
    field
      A : Obj
      α : Functor.F₀ F A ⇒ A
```

In the definition of `Bialgebra`, however, I needed to specify the *carrier* of the algebra. So I factored out the following definition:

```
F-Algebra-on : Obj → Set ℓ
F-Algebra-on A = F₀ A ⇒ A
```

And used in in the definition of `Bialgebra`:

```
record μ-Bialgebra (T F : Endofunctor C) (μ : DistributiveLaw T F)
   …
  field
    A : Obj
    a₁ : F-Algebra-on T A
    c₁ : F-Coalgebra-on F A
```

In converting between the different representations for the Bialgebras category however, I needed to also “get” `a₁/c₁` as (Co)Algebras. Therefore I wrote the following code:

```
to-Algebra : {A : Obj C} → {F : Endofunctor C} → (F-Algebra-on F A) → (F-Algebra F)
to-Algebra {A = A} α = record {A = A; α = α}
```

Since via the Curry-Howard correspondence dependent products encode existentials, what we are doing here is (un)wrapping existentials. I thought it an interesting language design question whether there could be a way to write datatypes such that this (un)wrapping doesn't require extra (boilerplate) code.

3.2 Dualizing proofs from `F-Algebras` for `F-Coalgebras`

See `Categories.Category.Construction.F-CoAlgebras`

For my formalization I needed to define the category of $\mathsf{Coalgs}(F)$. Now I didn't want to make it a "verbatim dual" (as these are called in comments in the library) of `Categories.Category.Construction.F-Algebras`. We take a verbatim dual to mean essentially a code clone of a dual construction except for perhaps compositions switched around in proofs. However, the datatype for Coalgebra (Morphisms) was *already* a verbatim dual of that for Algebras. So the only place I could still prevent code clones was in the proofs. I did this by writing suitable conversion functions to use proofs from $\mathsf{Algs}(F^{\mathrm{op}})$.

```
F-Coalgebras {C = C} F = record
  { Obj       = F-Coalgebra F
  ; _⇒_       = F-Coalgebra-Morphism
    …
  ; _∘_       = λ α₁ α₂ → record
    { f = f α₁ ∘ f α₂
    ; commutes = F-Algebra-Morphism.commutes
       (F-Coalgebra-Morphism⇒coF-Algebra-Morphism α₁
       coF-Algebras.∘
       F-Coalgebra-Morphism⇒coF-Algebra-Morphism α₂)
    }
  ; id        = record { f = id; commutes =
       F-Algebra-Morphism.commutes coF-Algebras.id }
    …
```

3.3 Elegant proof for `LiftAlgebras`

See `Categories.Functor.Construction.LiftAlgebras`

We have seen that we can lift F via μ to the category of $\mathsf{Algs}(T)$, giving us the functor: $\widehat{F}\colon \mathsf{Algs}(T) \to \mathsf{Algs}(T)$. The interesting part of constructing this functor is proving that $F_1 f$ is a T-Algebra-Morphism. The proof obligation is the following: Let $f\colon (X,x) \to (Y,y)$ be a T-Algebra-Morphism. Show $\widehat{F}f\colon \widehat{F}X \to \widehat{F}Y$ is also a T-Algebra-Morphism. That is, the following diagram commutes:

$$\begin{tikzcd}
TFX \arrow[r, "TFf"] \arrow[d, "\mu_X"] & TFY \arrow[d, "\mu_Y"] \\
FTX \arrow[r, "FTf"] \arrow[d, "Fx"] & FTY \arrow[d, "Fy"] \\
FX \arrow[r, "Ff"] & FY
\end{tikzcd}$$

This can be shown quite elegantly and succinctly with composition of commuting squares, by noticing that the upper is the naturality square of μ for f while the lower is the Algebra-Morphism square for f, lifted via F. I originally provided a more involved step-by-step proof:

```
commut {X} {Y} a = begin
  (F .F₁) (a .f) ∘ ((F .F₁) (X .α) ∘ (μ .η) (X .A))
  ≈⟨ pullˡ (⇔ (homomorphism F)) ⟩
  (F .F₁) ((a .f) ∘ (X .α)) ∘ (μ .η) (X .A)
  ≈⟨ ∘-resp-≈ˡ (F-resp-≈ F (commutes a)) ⟩
  (F .F₁) ((Y .α) ∘ ((T .F₁) (a .f))) ∘ (μ .η) (X .A)
  ≈⟨ ∘-resp-≈ˡ (homomorphism F) ⟩
  ((F .F₁) (Y .α) ∘ ((F .F₁) ((T .F₁) (a .f)))) ∘ (μ .η) (X .A)
  ≈⟨ pullʳ (sym-commute μ (f a)) ⟩
  (F .F₁) (Y .α) ∘ (μ .η) (Y .A) ∘ (T .F₁) ((F .F₁) (a .f))
  ≈⟨ sym-assoc ⟩
  ((F .F₁) (Y .α) ∘ (μ .η) (Y .A)) ∘ (T .F₁) ((F .F₁) (a .f)) ∎
```

I subsequently found a combinator in the library for composing commuting squares, and was able to use that to provide an elegant one-line proof:

```
commut {X} {Y} a = ⇔ (glue (⇔ ([ F ]-resp-square (commutes a))) (commute μ (f a)))
```

It should be noted however that it took quite some trial and error to arrive at the precise way the diagrams needed to be composed, and originally I had many `let`-expressions for subterms in my code to check that they had the types I expected. This was quite tedious to do and the first time that I got the impression that Agda can perhaps be called a proof *checker*, but not so much a proof *assistant*.

3.4 Lifting initial/terminal objects

Initial objects in $\mathsf{Algs}(T)$ may be lifted to $\mathsf{Coalgs}(\hat{F})$, and terminal objects in $\mathsf{Coalgs}(F)$ to $\mathsf{Algs}(\hat{T})$. We will consider the proof for lifted initials in detail, then discuss the dual proof of lifted terminals.

We lift $\bot$ to $\llparenthesis \hat{F}_0 \bot \rrparenthesis$. For some target object $\langle\langle X, a\rangle, c\rangle$ we define the initial morphism as $\llparenthesis a \rrparenthesis$. It remains to be shown that this T-Algebra-Morphism is in fact an $\hat{F}$-Coalgebra-Morphism, i.e. that $\hat{F}\llparenthesis a \rrparenthesis \circ \llparenthesis \hat{F}\bot \rrparenthesis = c \circ \llparenthesis a \rrparenthesis$. This is the case because both compositions are T-Algebra-Morphisms from the initial object with the same target (in the `agda-categories` library this lemma is called `!-unique₂`). The below diagram, where arrows are maps in the base category $\mathcal{C}$ shows all the involved objects and arrows:

$$\begin{tikzcd}
T\bot \arrow[rrr, dashed, "T\llparenthesis a \rrparenthesis"] \arrow[dr, "\mathrm{in}"] \arrow[ddd, dashed, "T\llparenthesis \hat{F}\bot \rrparenthesis"'] & & & TX \arrow[dl, "a"'] \arrow[ddd, "Tc"] \\
& \bot \arrow[r, dashed, "\llparenthesis a \rrparenthesis"] \arrow[d, dashed, "\llparenthesis \hat{F}\bot \rrparenthesis"] & X \arrow[d, "c"] & \\
& F\bot \arrow[r, dashed, "(\hat{F}=F)\llparenthesis a \rrparenthesis"'] & FX & \\
TF\bot \arrow[ur, "\hat{F}\mathrm{in}"] \arrow[rrr, dashed, "T\hat{F}\llparenthesis a \rrparenthesis"'] & & & TFX \arrow[ul, "\hat{F}a"']
\end{tikzcd}$$

Below is the corresponding agda-code:

```
liftInitial : Initial (F-Algebras T) → Initial (F-Coalgebras LiftAlgebras)
liftInitial μT = record
  { ⊥ = record
    { A = ⊥
    ; α = ⦅ F₀ ⊥ ⦆
    }
  ; ⊥-is-initial = record
    { ! = λ {A = X} →
      let
        a = F-Coalgebra.A X
        c = F-Coalgebra.α X
      in record
      { f = ⦅ a ⦆
      ; commutes = !-unique₂ (c ∘ ⦅ a ⦆) (F₁ ⦅ a ⦆ ∘ ⦅ F₀ ⊥ ⦆)
      }
    ; !-unique = λ {A = X} g → !-unique (F-Coalgebra-Morphism.f g)
    }
  }
  where
    open Initial μT
    open Category (F-Algebras T)
    open Definitions (F-Algebras T)
    open MR (F-Algebras T)
    open HomReasoning
    open Equiv
    private
      ⦅_⦆ = λ X → ! {A = X} -- "banana brackets" (Meijer 1991)
    open Functor LiftAlgebras
```

Note that we can be quite concise because all reasoning happens in $\mathsf{Algs}(T)$.

I initially tried to dualize the above proof for the dual theorem, arguing that, morally, `LiftCoalgebras T F μ ≅ (LiftAlgebras (Functor.op F) (Functor.op T) μ-op)`. This (modulo some conversion code I talked about in Section 3.2) was how I originally[2] had defined `LiftCoalgebras`. However, it turned out that dualizing `liftInitial` seemed to require a *hopeless* amount of glue code[3]. Therefore I gave a “verbatim” dual instead.

[2] see `LiftCoalgebras` at a9abd67

[3] see `F-Coalgebras` at 5dbdd6a for some of the (finally: discarded) glue code

Bridge

The lemma we are finally trying to prove is that the Bialgebra-maps from a lifted initial T-Algebra to a lifted terminal F-Coalgebra (byititiality and finality, respectively) are equal [2, p. 73]. In the previous section we got lifted initials in $\mathsf{Coalgs}(\hat{F})$ and terminals in $\mathsf{Algs}(\hat{T})$, respectively. In order to talk about morphisms between them however, we need to work in the same category. Therefore, in the following we prove $\mathsf{Coalgs}(\hat{F}) \cong \mathsf{Algs}(\hat{T})$.

3.5 Equivalences $\mathsf{Bialgs}(\mu, F, T) \cong \mathsf{Coalgs}(\hat{F}) \cong \mathsf{Algs}(\hat{T})$

See `Categories.Category.Construction.mu-Bialgebras`

I started out my formalization by defining the category of Bialgebras "on foot". However it turns out that in practice you'll want to use either one of the two representations $\mathsf{Coalgs}(\hat{F}) \cong \mathsf{Algs}(\hat{T})$ equivalent to it, as you are then reasoning in the (Co)Algebra category instead of in the underlying category $\mathcal{C}$. Since depending on what we are doing one or the other representation is more convenient, we need a way to convert between the two. Enter *Equivalences of Categories.*

We say two categories $\mathcal{C}, \mathcal{D}$ are *equivalent* if there exist functors $F\colon \mathcal{C} \to \mathcal{D}$, $G\colon \mathcal{D} \to \mathcal{C}$ such that $F \circ G \simeq 1_{\mathcal{D}}$, and $G \circ F \simeq 1_{\mathcal{C}}$, where $1_{.}$ is the Identity functor for a given category and $\cdot \simeq \cdot$ is a *natural isomorphism* between functors.

There was essentially only a single interesting detail, in the conversion $\mathsf{Coalgs}(\hat{F}) \Leftrightarrow \mathsf{Algs}(\hat{T})$, all the rest just came down to using the right (fields of) identities.

3.5.1 Converting between objects $\mathsf{Coalgs}(\hat{F}) \Leftrightarrow \mathsf{Algs}(\hat{T})$

Consider below objects in the respective categories. The (Co)Algebras have been colored red while the (Co)Algebra-Morphisms are black. Notice that in the the two diagrams, the $\mathcal{C}$-morphisms (a, c) switch roles.

$$\begin{tikzcd}
TX \arrow[r, "Tc"] \arrow[dd, "a"] & TFX \arrow[d, "\mu_X"] \\
 & FTX \arrow[d, "Fa"] \\
X \arrow[r, "c"] & FX
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[dd, "c"] & TX \arrow[l, "a"'] \arrow[d, "Tc"] \\
 & TFX \arrow[d, "\mu_X"] \\
FX & FTX \arrow[l, "Fa"']
\end{tikzcd}$$

As a diagram the right one can be produced by rotating the left one clockwise by one arrow. This seems simple enough, but this is because we have flattened everything down to the underlying category $\mathcal{C}$. The corresponding transformation in agda code (annoted with `(… = a/c)` to reconnect it to the above diagrams) is the following:

```
F₀ = λ X → record
  { A = to-Coalgebra $ (F-Algebra-Morphism.f $ F-Coalgebra.α X = c)
  ; α = record
    { f = (F-Algebra.α $ F-Coalgebra.A X = a)
    ; commutes = F-Algebra-Morphism.commutes (F-Coalgebra.α X) ∘ assoc
    }
  }
```

I call particular attention to the use of `○ assoc` in the proof: The obligation that $c \circ a = Fa \circ \hat{T}c = Fa \circ (\mu_X \circ Tc)$ can be produced from the premise that $c \circ a = \hat{F}a \circ Tc = (Fa \circ \mu_X) \circ Tc$ by transitivity of equality (`_○_`) and associativity of $\circ$ (`assoc`).

3.6 Final theorem

As mentioned at the end of Section 3.4, we prove that the Bialgebra-maps from a lifted initial T-Algebra to a lifted terminal F-Coalgebra (by it
itiality and finality, respectively) are equal [2, p. 73]. We obtain our lifted initial object in $\mathsf{Coalgs}(\hat{F})$ and our lifted terminal in $\mathsf{Algs}(\hat{T})$. We then need to conjugate the objects and morphisms in question to arrive at an equality in the same category (I chose to "transport" the lifted terminal object over into $\mathsf{Coalgs}(\hat{F})$, but the dual to that would be an equivalent option). For brevity's sake I have introduced the following abbreviations:

```
A2C = AlgebrasT̂⇒CoalgebrasF̂
C2A = CoalgebrasF̂⇒AlgebrasT̂
```

Then the final theorem states: `⦇ A2C₀ ⊤ ⦈ ≈ A2C₁ ⟦ C2A₀ ⊥ ⟧ ∘ id⇒A2C∘C2A ⊥`
The full agda code:

```
module _ (μT : Initial (F-Algebras T)) (νF : Terminal (F-Coalgebras F)) where
  open Functor
  open Initial (liftInitial T F μ μT)
  open Terminal (liftTerminal T F μ νF)
  open Category (F-Coalgebras (LiftAlgebras T F μ))
  open StrongEquivalence CoalgebrasF̂⇔AlgebrasT̂
  private
    module μT̂  = IsInitial ⊥-is-initial
    module νF̂  = IsTerminal ⊤-is-terminal
    A2C = AlgebrasT̂⇒CoalgebrasF̂
    C2A = CoalgebrasF̂⇒AlgebrasT̂
    id⇒A2C∘C2A : ∀ ( X : Obj ) → X ⇒ ((A2C ∘F C2A) .F₀ X)
    id⇒A2C∘C2A = G∘F≈id.⇐.η

  -- implicit args to `!` supplied here for clarity
  -- ⦇ A2C₀ ⊤ ⦈ ≈ A2C₁ ⟦ C2A₀ ⊥ ⟧ ∘ id⇒A2C∘C2A ⊥
  centralTheorem : μT̂ .! {A = A2C .F₀ ⊤} ≈ A2C. F₁ (νF̂ .! {A = C2A .F₀ ⊥}) ∘ id⇒A2C∘C2A ⊥
  centralTheorem = μT̂ .!-unique (A2C. F₁ νF̂ .! ∘ id⇒A2C∘C2A ⊥)
```

Below we present the theorem (modulo equivalences of categories, and where $\top, \bot$ refer are the non-lifted (co)initial objects) in diagrammatic form. With $\star$ we refer to $(\!|[\![\hat{T}\top]\!]|\!)$ / $[\![(\!|\hat{F}\bot|\!)]\!]$.

$$\begin{tikzcd}
T\bot \arrow[rrr, dashed, "T\star"] \arrow[dr, "\mathrm{in}"] \arrow[ddd, dashed, "T\llparenthesis \hat{F}\bot \rrparenthesis"] & & & T\top \arrow[dl, dashed, "\llbracket \hat{T}\top \rrbracket"] \arrow[ddd, "T\mathrm{out}"] \\
& \bot \arrow[r, dashed, "\llparenthesis \llbracket \hat{T}\top \rrbracket \rrparenthesis", "\llbracket \llparenthesis \hat{F}\bot \rrparenthesis \rrbracket"'] \arrow[d, dashed, "\llparenthesis \hat{F}\bot \rrparenthesis"] & \top \arrow[d, "\mathrm{out}"] & \\
& F\bot \arrow[r, dashed, "F\star"'] & F\top & \\
TF\bot \arrow[rrr, dashed, "TF\star"'] \arrow[ur, "\hat{F}\mathrm{in}"] & & & TF\top \arrow[ul, dashed, "F\llbracket \hat{T}\top \rrbracket \circ \mu_\top"']
\end{tikzcd}$$

4 What I learned about working with Agda

As mentioned at the end of Section 3.3, working with agda felt more like interacting with a proof *checker* than a proof assistant. If I was already confident of what a proof term should be, I was able to provide it and Agda would confirm that it fit. Sometimes the error messages helped me discover the issue was (s.a. that a term needed to be re-associated), but this was no simple feat. In general, having had experience with proof assistants such as Coq or Isabelle I missed amenities that allow more direct manipulations of the proof state such as `simplify in`, and interactive commands such as `Check` or `Search`.

5 Acknowledgements

I interacted with **Nathan van Doorn** and **Jacques Carette** on the Agda zulip channel from the very beginning. When it became clear I was going to contribute my code to the `agda-categories` library, **Jacques Carette** provided more in-depth feedback via review comments on my pull request.
One issue I ran into during the formalisation was *unsolved Metavariables* arising. In one instance this happens when some implicit arguments cannot be inferred. I spent quite some time debugging such an instance and in response submitted an issue to the Agda bug tracker. **Amélia Liao** submitted a fix making error messages in such instances more helpful, so I consider this a useful contribution to the community.
Originally I envisaged to possibly actually prove the correctness of a sorting algorithm defined following [2]. However, after discussions with **David Binder** and **Nathan van Doorn**, both of whom generously shared the incomplete Agda implementations of their previous attempts at similar formalizations, I decided to stick to implementing only the theoretical framework.
I would like to thank **Niels van der Weide** for their code review and suggestions, and **Jurriaan Rot** for suggestions on what to formalize and making me aware of "On Generalised Coinduction and Probabilistic Specification Formats" [1] and "Bialgebras for structural operational semantics: An introduction" [4].
Finally I would like to thank **Ruben Turkenburg** for being there to share my enthu-

siasm when I made progress in the formalization, and for input on using the `tikzcd` package.

References

[1] Falk Bartels. “On Generalised Coinduction and Probabilistic Specification Formats. Distributive laws in coalgebraic modelling”. PhD Dissertation. Vrije Universiteit Amsterdam, 2004.

[2] Ralf Hinze et al. “Sorting with bialgebras and distributive laws”. In: *WGP@ICFP*. 2012, pp. 69–80.

[3] Jason Z. S. Hu and Jacques Carette. “Formalizing category theory in Agda”. In: *CPP*. 2021, pp. 327–342.

[4] Bartek Klin. “Bialgebras for structural operational semantics: An introduction”. In: *Theor. Comput. Sci.* 412.38 (2011), pp. 5043–5069.

[5] Grant Reynold Malcolm. “Algebraic Data Types and Program Transformation”. English. PhD thesis. University of Groningen, 1990.

[6] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: *FPCA*. Vol. 523. 1991, pp. 124–144.

[7] Jan J. M. M. Rutten. “Universal coalgebra: a theory of systems”. In: *Theor. Comput. Sci.* 249.1 (2000), pp. 3–80.

[8] Daniele Turi and Gordon D. Plotkin. “Towards a Mathematical Operational Semantics”. In: *LICS*. 1997, pp. 280–291.

6 Reflection

During my research internship, I worked on formalizing the theory of bialgebras and distributive laws in Agda. Although I had no previous experience in programming and proving in Agda, I was able to learn it relatively quickly. One of the challenges that I encountered was the suboptimal ergonomics of theorem proving. Nevertheless, I was able to formalize a nice, self-contained subset of definitions and proofs. However, I only formalized simple distributive laws and did not instantiate the theorems as they were used in [2] in the analysis of sorting algorithms.

On the practical side, I was provided with a workplace in an office which I shared with PhD students at the chair. Next to the fact that this was a very nice working situation for me, it also provided the opportunity to discuss my research with other people at the chair, at the weekly seminar or during lunch/coffee breaks, notably with Niels van der Weide. Next to the offline interactions, I was able to interact online with members of the Agda community on the Agda Zulip and on my pull request on Github.

Looking ahead to my Master thesis and my future career, I want to continue exploring the applications of category theory to algorithm and programming language analysis and design. Furthermore, I want to expand my categorical toolbox and be able to recognize and apply more theorems, definitions, and constructions from Category Theory.

For my Master thesis, I would like to stay within the university and continue analyzing "sorting" categorically, although (also in the service of time and rapid prototyping) I no longer plan on formalizing my definitions and proofs.

Overall, my research internship was a valuable experience, as I was able to learn a new programming language and formalize/prove some interesting constructions/theorems. This experience has prepared me for future research and academic work, and I am looking forward to applying these skills to future projects.