Lecture 16: Gluing & Canonicity Continued, Natural Numbers

Lecturer: Max S. New
Scribe: Pranav Srinivasan

March 13, 2023

Abstract

We conclude discussion of the the gluing CT Structure by seeing how to represent the coproduct + function type structures in $\mathcal{G}$. Then, we begin discussion of how to encode infinite datatypes in STT.

1 $\mathcal{G}$ Constructions, Continued

1.1 Coproduct Structure in $\mathcal{G}$

We first show the construction for the sum type in $\mathcal{G}$, being careful to construct it so that syntactic projection (extracting the *ty* fields) preserves the sum types. For $\widehat{A}$ and $\widehat{B}$ in $\mathcal{G}_{ty}$, we define the 3 necessary components of $\widehat{A}\widehat{+}\widehat{B}$. Recall our definition of disjoint union of sets $X \uplus Y = \{(1,x) \mid x \ \in X\} \cup \{(2,y) \mid y \in Y\}$.

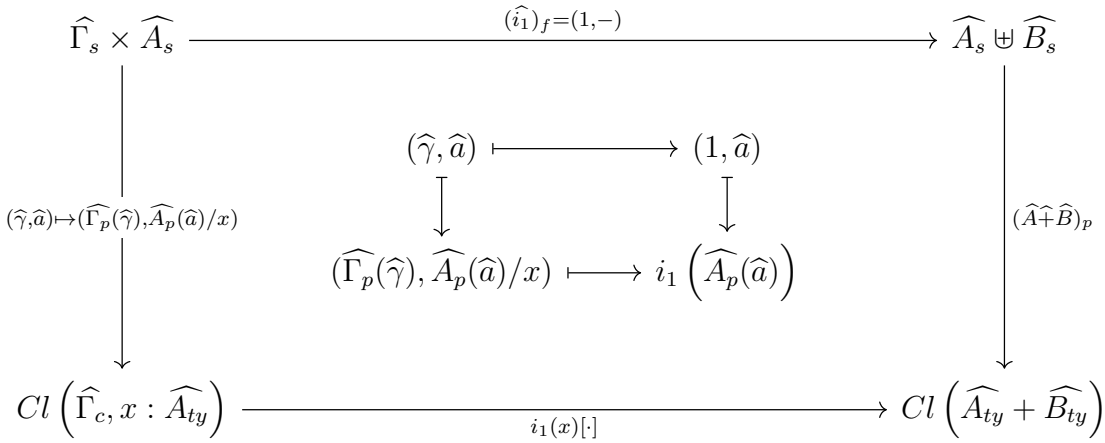
$$\widehat{A}\widehat{+}\widehat{B} := \begin{cases} (\widehat{A}\widehat{+}\widehat{B})_s & = \widehat{A}_s \uplus \widehat{B}_s \\ (\widehat{A}\widehat{+}\widehat{B})_{ty} & = \widehat{A}_{ty} + \widehat{B}_{ty} \\ (\widehat{A}\widehat{+}\widehat{B})_p(n,x) & = \begin{cases} i_1(\widehat{A}_p(x)) & n = 1 \\ i_2(\widehat{B}_p(x)) & n = 2 \end{cases} \end{cases}$$

Effectively, we propagate our sum constructs from Set and the underlying types, and have a function that does case matching to produce a term for a given element in the disjoint union set.

To verify it is a sum type, we check that it has a coproduct structure in $\mathrm{Un}_{\widehat{\Gamma}}$ for any $\widehat{\Gamma} \in \mathcal{G}_c$. For this, we define: $\widehat{i_1} \in \mathrm{Tm}_{\mathcal{G}}(\widehat{A}\widehat{+}\widehat{B})(\widehat{\Gamma}\widehat{\times}\widehat{\mathrm{sole}}\widehat{A})$ and $\widehat{i_2} \in \mathrm{Tm}_{\mathcal{G}}(\widehat{A}\widehat{+}\widehat{B})(\widehat{\Gamma}\widehat{\times}\widehat{\mathrm{sole}}\widehat{B})$:

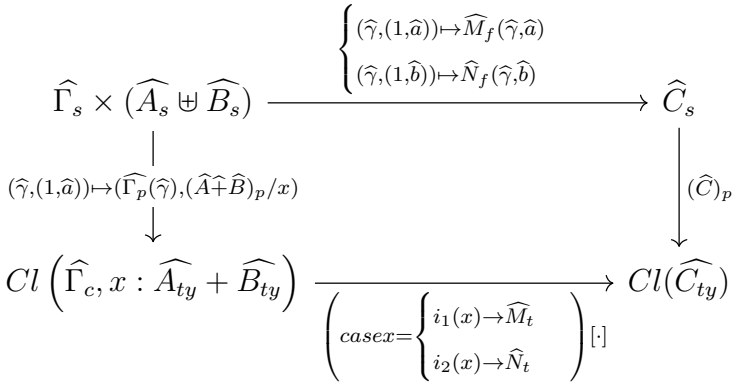
$$\widehat{i_1} = \begin{cases} (\widehat{i_1})_f : \widehat{\Gamma}_s \times \widehat{A_s} \to \widehat{A}_s \uplus \widehat{B}_s \\ (\widehat{i_1})_f(\widehat{\gamma}, \widehat{a}) := (1, \widehat{a}) \\ (\widehat{i_1})_t = \widehat{\Gamma}_c, x : \widehat{A}_{ty} \vdash i_1(x) : \widehat{A}_{ty} + \widehat{B}_{ty} \end{cases}$$

$\widehat{i_2}$ is defined analogously. We firstly need to verify that this term we've defined is valid, meaning it satisfies the commuting diagram regarding the actions of semantic evaluation and closing substitution.

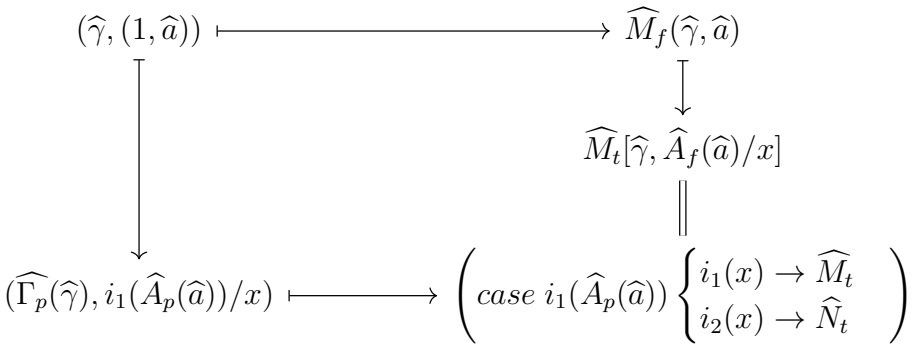


Now equipped with injections, we verify the coproduct structure. Assume we have another candidate coproduct $\widehat{C}$ and terms $\widehat{M} \in \mathrm{Tm}_{\mathcal{G}}(\widehat{C})(\widehat{\Gamma}\widehat{\times}\widehat{\mathrm{sole}}\widehat{A})$ and $\widehat{N} \in \mathrm{Tm}_{\mathcal{G}}(\widehat{C})(\widehat{\Gamma}\widehat{\times}\widehat{\mathrm{sole}}\widehat{B})$. We need then a unique term case $\widehat{M}\ \widehat{N} \in \mathrm{Tm}_{\mathcal{G}}(\widehat{C})(\widehat{\Gamma}\widehat{\times}\widehat{\mathrm{sole}}(\widehat{A}\widehat{+}\widehat{B}))$.

This is intuitively exactly the term you would expect: case-matching on the sum type and producing M or N accordingly. This is implemented both as a function of sets and as a syntactic function, and we verify that the diagram for performing the syntactic action after finding the associated term vs performing the semantic function and then finding the associated term produce the same result.



We do this by case work on the element of the disjoint union. We show the case for elements of the form $(1, \widehat{a})$.



Of note is the subtlety that the 2 paths don't give terms that are syntactically equal, but equal modulo the equational reasoning principles. This is acceptable as we are actually tracking equivalence classes of terms, and thus this diagram still commutes.

The fact that this construction makes the diagram commute is an argument that can be made at both the underlying set and type levels, thus guaranteeing the full diagram commutes. Similarly, uniqueness follows from the fact that we have componentwise uniqueness following from the underlying glued structures.

1.2 Function Type Structure in $\mathcal{G}$

We now present the similar structure for the function type, while omitting the details of verifying the construction is valid and satisfies the exponential structure. We can construct a first attempt:

$$\widehat{A} \widehat{\Rightarrow} \widehat{B} := \begin{cases} (\widehat{A} \widehat{\Rightarrow} \widehat{B})_s & = \mathrm{Set}(\widehat{A}_s, \widehat{B}_s) \\ (\widehat{A} \widehat{\Rightarrow} \widehat{B})_{ty} & = \widehat{A}_{ty} \Rightarrow \widehat{B}_{ty} \\ (\widehat{A} \widehat{\Rightarrow} \widehat{B})_p(f) & =? \end{cases}$$

That is, we simply let the glued set be the set of functions from the associated sets for $\widehat{A}$ and $\widehat{B}$. However, we run into problem when trying to define a function that consumes an arbitrary function and produces a closed term. Namely, not every function in $\mathrm{Set}(\widehat{A}_s, \widehat{B}_s)$ is computable, i.e. has an associated term in STT.

Thus, we want to restrict the associated set to functions that have an associated term that is semantically equivalent. Note, however, that this is precisely the defining characteristic of $\mathrm{Tm}_{\mathcal{G}}(\widehat{B})(\widehat{\mathrm{sole}A})$, i.e. functions that have a commuting diagram saying the associated function can be converted to the syntax and then substituted as an equivalent representation. Writing this slickly, we can say that

$$(\widehat{A} \widehat{\Rightarrow} \widehat{B})_s = \mathrm{Un.}(\widehat{A}, \widehat{B})$$

That is, the terms under the context $\cdot, x : \widehat{A}_{ty}$ with type $\widehat{B}_{ty}$, coupled with the associated set-theoretic function that the term defines.

Remark: This computability restriction is why this construction is sometimes referred to as Tait's Method of Computability.

With this additional restriction placed, we can define $(\widehat{A \Rightarrow B})_p$:

$$\begin{array}{ccc} \mathrm{Tm}_{\mathcal{G}}(\widehat{B})(\widehat{\mathrm{sole}A}) & \dashrightarrow & Cl(\widehat{A}_{ty} \Rightarrow \widehat{B}_{ty}) \\ \downarrow{\scriptstyle \pi_{\mathcal{L}}} & \nearrow_{\lambda} & \\ \mathrm{Tm}_{\mathcal{L}}(\widehat{B}_{ty})(x : \widehat{A}_{ty}) & & \end{array}$$

The intuition is that we simply project syntactically, recovering the underlying term under the context $x : \widehat{A}_{ty} \vdash M : \widehat{B}_{ty}$. From this term, we can construct a closed term of the function type, namely $\cdot \vdash \lambda x.M : (\widehat{A}_{ty} \Rightarrow \widehat{B}_{ty})$. Application and its universal property are left as an exercise.

2 Expanding STT to Infinite Datatypes

One limitation in STT as presented so far is its ability to reason about infinite datatypes, e.g. $\mathbb{N}$, lists & trees. In other words, considering $[\![\cdot]\!] : \mathcal{L} \to \mathrm{Set}$, it is not hard to check that as presented thus far, if every base type is interpreted as a finite set, then every set that this denotation specifies is finite. This is because every connective is interpreted as an operation (cartesian product, disjoint union, set of functions) that preserves finiteness.

We would like to extend STT with the ability to reason about infinite types. We present the construction for $\mathbb{N}$.

2.1 Naive Implementation of $\mathbb{N}$

A first attempt can be seen as a generalization of how we constructed other finite sets. We made the boolean type by adding the unit type 2 times $(1 + 1)$, and so, we can define $\mathbb{N}$ as

$$\underbrace{1 + 1 + \ldots}_{\mathbb{N}}$$

This gives us $\mathbb{N}$ many injection maps $i_n : 1 \to \mathbb{N}$. We can get pretty far with this approach, defining an introduction rule for each natural number

$$\frac{\Gamma \vdash M : 1}{\Gamma \vdash i_n(M) : \mathbb{N}}\ \mathbb{N}\mathrm{I}_n$$

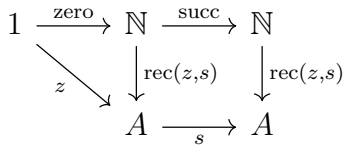
where we can think of $i_n(M)$ as n. However, this leads to a syntactic issue when specifying the elimination rule, namely, when we need to recover the underlying number, we need to pattern match against infinitely many cases:

$$\frac{\Gamma \vdash M_n : \mathbb{N} \quad \Gamma, x : 1 \vdash M_0 : A \quad \Gamma, x : 1 \vdash M_1 : A \quad \ldots}{\Gamma \vdash \left(\mathrm{case}_{\mathbb{N}} M_n \begin{cases} i_0(x) & \to M_0 \\ i_1(x) & \to M_1 \\ \ldots & \end{cases} \right) : A}\ \mathbb{N}\mathrm{E}$$

Thus, we aren't able to finitely express in this syntax how the following construction should be used. For this to usefully describe a programming language, we need a finitely expressible method of representing the universal property of the natural numbers.

2.2 The Natural Numbers Object & Primitive Recursion

We first lay out the property in Set, and then generalize to the categorical setting. The defining feature of $\mathbb{N}$ is the existence of zero (can be thought of as an element in $\mathbb{N}$, or equivalently as a morphism from the terminal object into $\mathbb{N}$), as well as an endofunction succ such that for any other object A with this construction, we can define a unique map $\text{rec}(z, s)$ that makes the following diagram commute.



That is, we need that

- $\text{rec}_{\mathbb{N}}(z, s)(\text{zero}) = z$
- $\text{rec}_{\mathbb{N}}(z, s)(\text{succ}(n)) = s(\text{rec}_{\mathbb{N}}(z, s)(n)$

The fact that this is a unique morphism allows us to recover the inductive proof technique and primitive recursion. That is, $f(\text{zero}) = z$ and for all $n \in \mathbb{N}$, $f(\text{succ}(n)) = s(f(n))$, then $f = \text{rec}_{\mathbb{N}}(z, s)(n)$.

We generalize this construction to a category $\mathcal{C}$ with a terminal object, and construct a **Natural Numbers Object**. N is a natural numbers object such that

- There exist morphisms $\text{zero} \in \mathcal{C}_1(1, N)$ and $\text{succ} \in \mathcal{C}_1(N, N)$ such that:
- Forall $A \in \mathcal{C}, z \in \mathcal{C}(1, A), s \in \mathcal{C}(A, A)$, there exists a unique $\text{rec}(z, s) \in \mathcal{C}(N, A)$ such that:
- $\text{rec}(z, s) \circ \text{zero} = z$ and $\text{succ} \circ \text{rec}(z, s) = s \circ \text{rec}(z, s)$

This gives rise to a nice representation syntactically in STT as well. We now have 2 introduction rules:

$$\frac{}{\Gamma \vdash \text{zero} : \mathbb{N}}\,\mathbb{N}\text{I}_{\text{zero}} \qquad \frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \text{succ}(M) : \mathbb{N}}\,\mathbb{N}\text{I}_{\text{succ}}$$

and an elimination rule that lets us unroll one iteration of the recursively defined number:

$$\frac{\Gamma \vdash M : \mathbb{N} \quad \Gamma \vdash M_b : A \quad \Gamma, x : A \vdash M_s : A}{\Gamma \vdash \left(\text{rec}_{\mathbb{N}} M \begin{cases} base & \to M_b \\ step(x) & \to M_s \end{cases}\right) : A}\,\mathbb{N}\text{E}$$

The β rules define how this unrolling takes place, as well as provides a base case:

$$\frac{}{\Gamma \vdash \left(\mathrm{rec}_{\mathbb{N}}(\mathrm{zero}) \begin{cases} base & \to M_b \\ step(x) & \to M_s \end{cases}\right) = M_b : A}\mathbb{N}\beta_{\mathrm{zero}}$$

$$\frac{}{\Gamma \vdash \left(\mathrm{rec}_{\mathbb{N}}(\mathrm{succ(M)}) \begin{cases} base & \to M_b \\ step(x) & \to M_s \end{cases}\right) = M_s \left[\left(\mathrm{rec}_{\mathbb{N}}(\mathrm{M}) \begin{cases} base & \to M_b \\ step(x) & \to M_s \end{cases}\right)/x\right] : A}\mathbb{N}\beta_{\mathrm{succ}}$$

Finally, the η rule captures the "proof by induction" technique.

$$\frac{\Gamma \vdash M'[\mathrm{zero}/x] = M_b \quad \Gamma \vdash M'[\mathrm{succ(x)}/x] = M_s[M'/y]}{\Gamma \vdash M' = \left(\mathrm{rec}_{\mathbb{N}}(x) \begin{cases} base & \to M_b \\ step(y) & \to M_s \end{cases}\right)}\mathbb{N}\eta$$