The Way of Glue
An Invitation to the Categorical Semantics of Lambda Calculi

A Thesis
Presented to
The Established Interdisciplinary Committee for
Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Jay Kruer

December 2021

Approved for the Division
(Mathematics and Computer Science)

Angélica Osorno	James (Jim) Fix

Acknowledgements

It's not often in life one gets a chance to thank the many people who have helped him on his way, so I've spent the last few months slowly accumulating a list of people I'd like to thank whether for their direct impact in making this thesis possible or just for having had an outsized impact on my life at one point or another. I can only start by thanking my advisors Angélica and Jim for their part in making my thesis year one of my favorite's in my time at Reed and for making this thesis possible. I am first of all indebted to Angélica's spooky ability to instantly grasp at understanding even in an area relatively unknown to her. Without her hints at how to tackle an opaque construction every now and then, I think I'd probably be still fumbling around with the basics. Jim has had an outsized influence on the direction my intellectual life has taken. Jim's intro CS course taught partially in Standard ML was my first real exposure to mathematically structure programming and opened up to me a world that had until then had seemed inaccessible. Jim has also served as a great mentor in matters of personal taste and style (his course webpages are always out of this world cool.) I'll really miss our wacky math-cs crossover meetings, you two!

The students and faculty in the math department have made the last few years a profoundly enriching experience. Albyn Jones and Irena Swanson, who taught my intro courses freshman year, are singled out for special thanks. I was not by any stretch a likely math major by the time I came to Reed. Having been brought from complete ignorance to a great appreciation for mathematics under Albyn and Irena's careful guidance was a great privilege and one I'll remember fondly all my life! I am also grateful to Kyle Ormsby for always being an absolute chiller, encouraging me in some early ambitions, and for giving me a taste of category theory before I fled from his topology class.

Now the acknowledgements become less organized, but by no means less sincere!

None of this would have been possible without my Mom and Dad. Thanks so much!

Jaclyn, my love, thank you for being everything to me over the past (almost!) two years and for your friendship for the last (almost) 4. You turned an ugly quarantine into a beautiful one, and I can't wait to spend the rest of it (and more!) with you.

Thank you to Uncle Jay for your constant encouragement for my interests in computers. And to Grandma, Aunt Julie, Aunt Susie, and all of my extended family for your continued support!

A few teachers have made a huge impact on my life. My high school Chinese teacher 吳老師 (Rhoda Weston) is the most generous person I've ever met, in so

many ways, and just about changed my life. Thank you for making it possible for me to go to Taiwan and for everything else! Thank you to my first grade teacher Ms. King for being a fantastic example of living with kindness, to 柳老師 (Hyong Rhew) for getting me to chill out a bit freshman year[1] and hours of fun studying Chinese, to Mrs. Leitsch for stellar teaching and encouragement in 9th grade English, to Mr. Cool for good times on the robotics team (even if you wouldn't let us switch from LabView), and to my 4th (?) grade history teacher Mr. Raveli for teaching me that it's better to know how and where to find the answers than to have them all committed to memory.

More than anything, friends have made the last few years special:

Thanks to the hyperbolic crew—Francis Baer, Alec Forget, Usman Hafiz, Kiana McBride, and Luke Doms—for helping to make Kyle's wild ride a dummy good time.

Thanks to the many friendbobs of pika: Joebob, Jit, Eli, Gabe and Ciara, Becca, Andres, Nathan, Chloe. The time I spent with you all in Boston was and remains precious to me beyond measure!

Thanks to Aaron Weiss for being a constant companion (even over discord) over the last 3 years. I'm so glad we ended up labmates! Stay fresh, king.

The following people made my freshman year at Reed. Thank you to Holden and Max for making WP117 home. Special thanks to Max for not holding against me my having thrown a mug at him, and to Holden for never violating the NAP. Thanks to Noah Koster for being one of my very first friends at Reed and for always truckin'. You may not be the next Bill Gates, but you're a great guy. Thanks to Nick Chaiyachakorn for always encouraging my stupid computer purchases, and for just always being there for me. Thanks to Yuta, Shulav, Jake, and Aditya for many fine hours of good lad time. Thanks to Carter Fife for ditching us all and for being a massive weeb, to Dan Jeon for telling everyone about that time he caught me practicing Naruto hand signs, to Mary Brady for showing me the best song of the century (Freaky Friday by Lil Dicky ft. Chris Brown), and to Salma Huque for being poggers.

The following people made my sophomore year much more tolerable. Thanks to Genya, Tristan, and the rest of the crew for so many great nights in Polytopia. Special thanks go to Young Kim for corrupting my soul. This thesis is in some sense a tribute to you, because I know how much you love category theory. Thanks also to Alice McKean for good times talkin' type theory.

Thank you to my beloved 1989 Toyota TownAce van which made the early pandemic an absolute adventure. I'll miss you forever :'(

My deep thanks go to Amal Ahmed for taking me on and guiding my early studies of programming language theory during my leave from Reed. I came to her with a lot of enthusiasm to learn and not much else, and she gave me a wonderful opportunity to make good on that enthusiasm. Without her guidance I would not be writing a thesis like this today, and I'll always be indebted to the influence she's had on my way of thinking about computing.

[1]Thanks also for comparing me to a metal waterbottle for some reason…never figured that one out.

Thanks to Murali Vijayaraghavan for mentorship and patient instruction while I was learning the ropes the ropes of CPU and hardware design. Thanks to Cody Roux and Dan McArdle (then at Draper Lab) for early encouragement!

Thank you to Sara Rosenberger in the business office for having my back when the office of financial aid left me to rot. This thesis couldn't have happened without you!

Thanks to my sister Megan for her unrelenting support, especially over the last few challenging years! and for her patience putting up with my various neuroses! Thanks to my brother Ryan and my sister Kathleen for always being there.

Thank you to the composers of Breath of the Wild and to lofi girl for the soundtrack.

I owe a huge debt to the great and generous people of the TEX StackExchange for TEXmacros used in the production of this document. Thank you to user4586 for their automatically-sized delimeters macro, to Steven B. Segletes for their really wide tildes macro, and to user13907 for their stylized B symbol. Thank you to the various people involved in the creation of the Lua LATEX system, in whatever capacity. Free software development is often thankless work, and I hope that recognizing it in this small way means something to the authors of these tools.

Thank you to Steve Awodey for listening to me deliver a draft version of the argument laid out in Chapter 3 and for helpful comments. Thanks to Carlo Angiuli for answering a question about the category of renamings. Thank you to Ivan Di Liberti for many fun and useful mentorship Zoom meetings throughout the last few months!

I would like to give special thanks to Paul Taylor who wrote the book that made this thesis possible. Paul's *Practical Foundations of Mathematics* is one of the great books of computing, in the same league as Knuth's *Art of Computer Programming* in that it is not just a useful centralization of human knowledge but that the presentation is painstakingly beautiful. On nearly all of the early days spent on this project I learned from Paul's book a profound perspective on programming or mathematics in general.

My thanks also go to Jon Erickson, author of *Hacking: The Art of Exploitation* which got me started on this road so many years ago. What a long and strange road to a thesis about huffing (strictly in the mathematical sense) glue!

Notation

The face $\mathfrak{Mathfrak}$ is used to denote presheaves, namely, objects in the category $\mathfrak{Set}^{C^{\mathrm{op}}}$ for any category C.

Table of Contents

Introduction **1**

Chapter 1: Functorial semantics **7**
1.1 An algebraic prelude 8
1.2 Elementary sketches and their models 9
1.3 The category of contexts and substitutions 12
1.4 Models are essentially functors 13
1.5 Algebraic theories and their algebras 14

Chapter 2: Syntax and (functorial) semantics of the lambda calculus **23**
2.1 The simply typed lambda calculus 23
2.2 Raw CCS and beta-eta is exactly CCS 24
2.3 An algebraic treatment of the lambda calculus 26

Chapter 3: Normalization by gluing **31**
3.1 The comma construction and friends 34
3.2 An easy instance of the comma construction: the category of renamings 35
3.3 Variable-arity Kripke relations 36
3.4 A harder comma category: the gluing category; or, Artin's monster . 37

Chapter 4: Obtaining a normalization function **41**
4.1 Presheaves of neutral and normal syntax 41
4.1.1 Variables and binding in Renhat 41
4.1.2 Stratifying neutral and normal terms 42
4.1.3 Presheaves of syntax over the category of renamings 44
4.1.4 A stratified lambda-algebra of neutrals and normals in Renhat 46
4.2 Desiderata for a normalization function 47
4.3 Cartesian closed structure for the gluing category 48
4.4 Gluing syntax to semantics 52
4.5 Taking stock: charting a path to normalization 56
4.6 Glued reification and reflection 58
4.7 A promise kept: a function from open terms to normal terms 61
4.8 Reaping what we've sown: easy proofs of the correctness properties . 62

Bibliography **65**

List of Tables

Abstract

Normalization by gluing is based.

Introduction

> And further, by these, my son, be admonished: of making many books there is no end; and much study is a weariness of the flesh.
>
> Ecclesiastes 12:12, KJV.

Though we encounter many interesting objects and ideas along the way, this thesis is primarily concerned with the *open normalization* problem for the simply typed lambda calculus. Some explaining is in order. First, we'll talk about the simply typed lambda calculus. The reader might be familiar with the *untyped* lambda calculus. The lambda calculus is a formal system oriented toward the study of functions and their *applicative* behavior [Bar12]. We define a notion of function abstraction which allows us to select *bound* variables from the variables appearing in some mathematical phrase, as in $\lambda x.\, x^2$. The fundamental operation of the lambda calculus is *application*, which is defined in terms of *substitution*. The application of a function f to an argument a is denoted by juxtaposition as in

$$fa.$$

A more concrete example of application is $\sin \pi$. Function application is a derived notion; what application *does* is defined in terms of *substitution*. For phrases (henceforth called *terms*) e and e', we impose an equation

$$(\lambda x.\, e)\, e' = \left[e'/x\right]^* e$$

called the *β-law*. When writing

$$\left[s/x\right]^* p,$$

we mean the term given by taking the term p and replacing with s every occurrence of x appearing within. Authors more interested in the computational character of the *lambda calculus* write

$$(\lambda x.\, e)\, e' \to \left[e'/x\right]^* e$$

so that β principle is instead understood as a *reduction rule* for an *allowed transition* or *computation step*. For our purposes, we will mostly be working with equations except in remarks meant to appeal to those readers who are operationally inclined.

The famed Church-Turing thesis says that the lambda calculus and the Turing machine, a more famous model of universal computation, are *equivalent in computational power*. That is, any Turing machine can be simulated by the lambda calculus

and conversely any program written in the lambda calculus can be simulated by some Turing machine. Consequently, just as the Turing machine is a universal computer, the lambda calculus is a universal programming language. That the lambda calculus is so powerful is exciting, but some consider working with that full power unwieldy or even dangerous. Indeed, there are programs that one can write down in the lambda calculus which get "stuck" before computing a *value*, meaning that there are lambda programs p which reduce to a p' which isn't a value but cannot be reduced to some p''.

The usual solution to this problem is to tie our own hands a bit, imposing a regime of *types* which allow us to statically rule out those program states (i.e., stuck terms) that we don't want to happen. The simplest example of such a regime is, appropriately, *simple types*. The simply-typed lambda calculus enriches the lambda calculus with some base types, say booleans and natural numbers, along with derived types $A \times B$ representing *pairs of terms* of the types A and B along with $A \Rightarrow B$ representing *functions* taking an A and giving a B. For example, the type $\mathbb{B} \times \mathbb{N}$ is inhabited by pairs of a boolean and some natural number. Along with this type structure comes *rules* stipulating how it may be used. We will not explain this story in depth in this thesis, but we will soon see these rules in Chapter 2. The desired result indeed follows from this setup: lambda terms that follow the typing rules do not get stuck when evaluated. This is one instance of the famed *type safety* properties of programming language theory. Another family of properties, namely *normalization* properties, follows from this type regime. It turns out that any program in the lambda calculus will reduce to a value in a finite number of evaluation steps. This property is called *closed normalization*. Borrowing the language of the theory of Turing machines, this means that programs written in the simply typed lambda calculus always halt. In particular, this means that the lambda calculus is *not* Turing complete and can only simulate deciding Turing machines. This can be a problem for some applications. If you want, say, your web server to run indefinitely without bound; the simply lambda calculus can't do that. You could, however, make your program loop until the projected death of the sun and deploy a new version of your server shortly thereafter. Having noted that there are some workarounds to the problems arising from the terminating character of the simply typed lambda calculus, it is interesting to note also that this very character is actually required for some applications; for some purposes, non-terminating languages have no such cute workarounds.

An *interactive theorem prover* is a computer program that allows a computer user to state theorems and input proofs[2], which are then checked by the program. Many interactive proof systems, such as Coq or Agda, are based on type[3] theory. Such proof systems leverage a principle called the *Curry-Howard isomorphism*, which allows one to represent theorems as *types* and proofs as *programs*. The advanced type theories of such systems allow one to state and prove the sort of universally and existentially

[2]Individual systems vary in how manual the process of inputting proofs is.

[3]Here I am referring to type theory in the same sense as above for the simply typed lambda calculs. Only the type systems underlying modern proof assistants feature much more complicated type structure, which is precisely what enables them to express non-trivial mathematical statements and proofs. We will see a little more on this story below.

quantified propositions one often encounters in mathematics. An *absolutely critical* property for systems like this is that the proof construction language (remember that this is analogous to a programming language according to the Curry-Howard isomorphism) be terminating, for the following reason. Suppose you have stated a proposition P. The Curry-Howard isomorphism allows us to represent P by a type T_P such that constructing a term (a program potentially with parameters) t with type T_P amounts to proving the proposition P. Given any type, I can give you a non-terminating program of that type. This is easiest to see by putting unguarded termination in the language as a form of non-termination. The program simply calls itself recursively forever, infinitely passing the buck but ostensibly having the desired type. Let's check out an example in the Coq proof assistant; we will be deriving a term of type False (the type which has no term constructors, and hence really ought to be uninhabited.) You could perform this same construction for any type you choose, even if it would be redundant now that we have a False term (which gives us a term of any other type by trivial case analysis). We first manually deactivate a Coq safety feature expressly intended to rule out such definitions.

```
j@art-house-thinking-sand ~ % coqtop
Welcome to Coq 8.14.0

Coq < Unset Guard Checking.

Coq < Fixpoint falso (u : unit) : False := falso tt.
falso is defined
falso is recursively defined (guarded on 1st argument)
```

If we reset *guard checking*, which places restrictions on the allowed forms of recursive definitions so that they are *well-founded*, we find that Coq refuses this definition.

```
Coq < Set Guard Checking.

Coq < Fixpoint falso (u : unit) : False := falso tt.
Toplevel input, characters 0-46:
> Fixpoint falso (u : unit) : False := falso tt.
> ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error:
Recursive definition of falso is ill-formed.
In environment
falso : unit -> False
u : unit
Recursive call to falso has principal argument equal to
"tt" instead of a subterm of "u".
Recursive definition is: "fun _ : unit => falso tt".
```

With this fact in mind, normalization proofs for the underlying proof language are an essential requirement for a trustworthy interactive theorem proving system;

otherwise there is no guarantee that proofs of false theorems as above are unavailable. It should be noted that non-termination is only one source of inconsistency and that we should demand much more than termination of our theorem proving systems.

Normalization theorems like these are typically proven using a technique called *the method of logical relations*[4] We will learn a bit more about logical relations in the third chapter. The normalization theorem of interest to people concerned with the safety of proof assistants are a little more challenging, and demand a leap in technology to the method of *Kripke logical relations*, also detailed in Chapter 3. The class of normalization theorems of interest to implementors of proof assistants is called the problem of *open normalization*, and is in fact the type of theorem we will work toward in this thesis. Our *open normalization* theorem will be stated somewhat differently than the *closed normalization* statement given above. Rather than speaking directly about the behavior of terms undergoing reduction, we will work with *equivalence classes* of terms taken over an equality *generated by* the reduction rules of the lambda calculus. Our normalization theorem will demand that we produce a *function* from ordinary open terms to normal forms which both preserves and reflects these equivalence classes of terms. The reason that such a device is interesting to implementors of proof assistants is that proof assistants are usually based on *dependent type systems*, in which the language of types allows for the presence of general terms. Such advanced type systems are the ones alluded to earlier in this introduction during our discussion of the Curry-Howard isomorphism. The Curry-Howard isomorphism casts the task of proof verification as the task of typechecking. Because dependent type theory allows for arbitrary terms (and in particular *programs*) inside of types, the task of typechecking in this setting amounts to deciding the equality of the final *result* of arbitrary subterms (and, in particular, arbitrary sub*programs*). In the general case, this problem is well-known to be undecidable. Using a *normalizing* term language makes it possible. Coming up with a *normalization* function nf_τ^Γ which satisfies some key properties[5] will allow us to compute normal forms for these subterms and use those normal forms for the desired equality check. The desired property most relevant to dependent typechecking is what I call *equation reflection* which says that for open terms t and t' of type τ whose free[6] variables come from *context*[7] Γ we have that

$$\mathrm{nf}_\tau^\Gamma(t) = \mathrm{nf}_\tau^\Gamma\left(t'\right) \Rightarrow t = t'.$$

This property validates the use of normalized subterms (as opposed to the original subterms) in our final check for equality of types. More information on how normalization functions can be used in implementations of dependent typecheckers can be found in Christiansen's tutorial [Chr].

Theorems such as closed and open normalization are conventionally proven using

[4]Normalization proofs actually only require the use of *logical predicates*, which are the unary case of logical relations, but the general method is better known by the name *logical relations*.

[5]Exactly which properties we will demand of our normalization function will be fully detailed in Chapter 4.

[6]Free variables are those variables appearing in a term which are not bound by a lambda abstraction.

[7]A context is nothing more than a list of variables together with their associated types.

a technique called the method of *logical relations*. Open normalization in particular demands the use of a strong variant of logical relations called *Kripke logical relations*. We will discuss Kripke logical relations in more detail in the introduction to Chapter 3. The specific open normalization theorem often proven using Kripke logical relations is somewhat ill-suited to the proof assistant applications mentioned in the previous chapter; the theorem statement only asks for a proof of *existence* of a normal form for each term, whereas dependent typechecking requires an effective formula by which to *compute* the normal form for a given open term. Our goal in this thesis is to construct such a normalization function. To do so, we make a significant departure from the usual methods. However, we will ultimately observe echoes of the conventional approaches in the constructions we settle on.

The methods used in this thesis come from category theory and in particular topos theory, but only basic familiarity[8] with category theory is assumed. We will also make use of many of the basic tools of programming language theory without much commentary. In particular we expect that the reader has a working knowledge of type systems, grammars, etc. For these essentials, we refer the reader to Pierce's textbook [Pie02]. I regret that I wasn't able to fully treat these topics in this thesis, but such a document would have been at least twice as long and I'm just one lad.

The study of this mountain of prerequisites is well worth it, as the benefits of the methods we employ are many. First of all, the *Artin gluing* construction which frames the development enables us to give a far more conceptual treatment of this problem than conventional approaches allow for. Our particular instance of the Artin gluing construction allows us to cast the normalization problem as a simpler problem of finding a a morphism from one chosen object to another chosen object inside of a specially structured category. The structure of the category is such that each intermediate step taken along the path from the first chosen object to the second chosen object must maintain a certain invariant which ultimately will entail our desired properties. One could liken this situation to the style of *type-directed programming* popular among functional programmers. In *type-directed programming*, one defines rich datatypes in terms of which the functions comprising one's program are *specified* before turning to the task of actually writing them. These (possibly dependent) datatypes are used to express a variety of correctness criteria about the functions such that any function the programmer ultimately comes us with meets the specified properties as long as it has the right type (as determined by the typechecker). In a similar way, the design of the gluing category is like a strongly typed setting in which to define and compose syntactic transformations. The structure of the gluing category forces us to verify a certain correctness criterion *as we go* while constructing the intermediate syntactic transformations so that, besides one final totally straightforward verification about each of the syntactic transformations, our desired correctness criteria for a normalization function immediately follow from having managed to construct a normalization

[8]The level of familiarity I expect of the reader is the kind that one might obtain from having read through most of a standard text such as Riehl's [Rie] or Awodey's [Awo10] or from having read the early parts of a text like Barr and Wells's [BW]. I recommend that the reader unfamiliar with category theory check out as many texts as possible, as each presents its own unique perspective and one perspective may suit your background better than another.

function of the "correct type" within the gluing category. I am remaining vague on this point for now because we don't yet have enough language to precisely describe what is going on. A more clear characterization of how the gluing category productively ties our hands as typed programming languages do shall come in Chapter 3. For now, we turn to developing category-theoretic tools for reasoning about syntactic theories and the meaning of the things (terms) these theories allow us to write down.

Chapter 1

Functorial semantics: sketches and their models; algebraic theories and their algebras

In this chapter we will develop tools for reasoning about (syntactic) *theories*, which are *notions of* abstract structure. As an example, this chapter will present a theory encoding the structure of a ring (as in algebra.) A more complicated example will come in Chapter 2 which will apply the ideas of this chapter to simple type theory. To discuss how we write down our theories, we need another level of abstraction. We will work with several *notions of* (syntactic) theory. We will more laconically refer to a notion of syntactic theory as a *doctrine*. A *doctrine* is something like a framework specifying how we are to write down a theory. The first doctrine we will consider is that of the *elementary sketch*[1]. The doctrine of the *elementary sketch* allows us to specify theories involving *unary* operations (those defined over a single parameter.) This restriction on the arity of operations turns out to be quite limiting. To address this, we will later upgrade the doctrine of elementary sketches to the doctrine of *algebraic theories* which allow us to encode operations with any finite number of parameters, thus covering a broad variety of gadgets one might encounter in mathematics and computing. Algebraic theories are known more famously as *Lawvere theories* after categorical logic superstar (and former Reed College professor!) William Lawvere who originally presented them while building his functorial treatment of universal algebra. Emphasizing the doctrinal upgrade, some authors refer to algebraic theories as *finite product sketches* [Wel09] though we will tend to stick with Lawvere's original terminology. As an example of the strength of algebraic theories, we will show how to write down (as an algebraic theory) what it means to be a ring, with no reference to sets or functions. In the following chapter, we will use the doctrine of algebraic theories to develop categorical semantics of the lambda calculus. This chapter is strictly expository in nature; indeed, much of what follows draws heavily from Paul Taylor's *Practical Foundations of Mathematics* [Tay99]. My contribution is

[1]The reader familiar with and/or traumatized by experience with pencils of geodesics in hyperbolic geometry need not be scared away by this terminology; there is nothing non-Euclidean afoot here, per se.

to flesh out some of the examples found there, add some of my own, and make other parts of the presentation more palatable and quickly digestible to the reader already acquainted with basic category theory and type theory.

1.1 An algebraic prelude

We begin by recalling from algebra the notion of an *action* of, say, a group or a monoid. Actions are, from our perspective, a way of giving meaning, or *semantics* to elements of a set which enjoys some algebraic structure.

Definition 1.1.1. Recall that a **covariant action** of a group or monoid $(M, id, \cdot)$ on a set A is a function $(-)_* (=) : M \times A \to A$ such that $\mathrm{id}_* a = a$ and $(g \circ f)_* a = g_* (f_* a)$. We can similarly define the notion of a **contravariant action** which similarly requires the identity action to do nothing, but instead flips the order of action for compositions: $(g \circ f)^* a = f^* (g^* a)$

For example, in algebra we learn that the dihedral group of size 8, written D_4, acts on the square (with uniquely identified points) by reflections and rotations; each of the operations encoded by the action results in the same image of the square up to ignoring the unique identity of the points we started with. This action gives geometric meaning to each of the group elements, and was used in the first day of the author's algebra class to explain the algebraic mechanics of the group itself; discovering which elements of the group are inverse to one another is done by geometric experimentation using a square with uniquely colored vertices. Similarly, the symmetric groups S_n act on lists of length n by permutation of the list elements. In this case, the action can be even more trivially defined. We now turn to the definition of an important property of actions: *faithfulness*.

Definition 1.1.2. A **faithful** action $(-)_*$ is one for which we have

$$\forall (a : A) \,.\, f_* a = g_* a \Longrightarrow f = g$$

for all f, g in M. Morally, this requirement says that elements are *semantically* equal (or, act the same) only when they are *syntactically* equal (or, *look* identical.)

It can now be seen that the crucial property enjoyed by the natural action of D_8 on the square which enabled our use of paper cutouts in studying the group is faithfulness. If the action were not faithful, determining which operations in D_8 are inverses would not be so easy as printing out a square and plugging away, because we may (among other catastrophes) be working with an action which may not take only the identity element to the leave-everything-in-place operation on the square.

Having gesticulated that actions gives groups and monoids their meaning, we turn to the development of the doctrines of *elementary sketch* and *algebraic theory* which will allow us to generalize both sets with algebraic structures and their actions to new settings.

1.2 Elementary sketches and their models

As promised, we begin with the definition.

Definition 1.2.1. An **elementary sketch** is comprised of the following data:

1. a collection $X, Y, Z, \ldots$ of named **base types** or **sorts**
2. a **variable** $x : X$ for each occurrence of each named sort.
3. a collection of **unary operation-symbols** or **constructors** τ having at most one variable. We write these operation symbols as, for example, $x : X \vdash \tau(x) : Y$.
4. a collection of equations or **laws** of the form:

$$\tau_n\left(\tau_{n-1}\left(\cdots\tau_2\left(\tau_1(x)\right)\cdots\right)\right) = \sigma_m\left(\sigma_{m-1}\left(\cdots\sigma_2\left(\sigma_1(x)\right)\cdots\right)\right)$$

We will discuss the generality provided by this definition after some intervening examples. One of the simplest examples is the sketch of a (free) monoid on some set S:

Example 1.2.2 (Sketch of a (free) monoid)**.** The requisite data for the sketch are as follows:

1. The collection of sorts is the singleton $\{M\}$.
2. The collection of variables is $\{m : M\}$.
3. The operation symbols are the set $\left\{m : M \vdash s(m) \mid s \in S\right\}$.
4. No equations are imposed.

To really buy that this sketch generates a free monoid, we need an intervening concept which makes more concrete what we mean when saying "generates." The idea should be familiar to the reader acquainted with type theory, despite the drastically simplified setting.

Definition 1.2.3. Given an elementary sketch, a **term** $t : \Gamma \vdash X$ is a string of composable unary operation-symbols applied to a variable $\gamma : \Gamma$ as in:

$$\tau_n\left(\tau_{n-1}\left(\cdots\left(\tau_2\left(\tau_1(\gamma)\right)\right)\right)\right).$$

Composable unary-operation symbols are ones which have compatible domains and codomains in the usual sense, as for example in set theory.

We now propose a more precise version of the above claim: the terms of the sketch defined above form the elements of a free monoid over S. Before we can continue, we should decide what our term composition will be.

Definition 1.2.4 (Composition of terms)**.** Composition of terms is by substitution for the variable: for a term $s : \Delta \vdash \Gamma$ and some terms t_i with t_n having as its outermost operation symbol an operation ending in Ξ, we define

$$\left(t_n\left(t_{n-1}\left(\cdots\left(t_2\left(t_1\left(\gamma\right)\right)\right)\right)\right)\right) \circ s = t_n\left(t_{n-1}\left(\cdots\left(t_2\left(t_1\left(s\left(\delta\right)\right)\right)\right)\right)\right) : \Delta \vdash \Xi.$$

With our notion of composition in hand, we can now handwave an argument for our revised claim that the terms of the sketch form the elements of a free monoid. It is well-known that substitution operation is associative; see, for example, Chapter 1 of [Tay99] for a fantastic treatment of the theory of substitution. As a consequence, any elementary sketch, including this one, satisfies the associativity axiom of monoids for free. The identity term is given by zero composable unary operation-symbols (i.e., the empty string) applied to a variable. Because composition is given by substitution for the variable, a lone variable does indeed work function as the identity. Freedom comes from the fact that the sketch has no equations.

This loose argument is somewhat satisfying, but we can do better. To get there, we will first develop a notion generalizing *actions* from algebra. After doing so, we will give more concrete meaning to this sketch and complete our intuitive handle on it.

Definition 1.2.5. A **model** (also known as an algebra, an interpretation, a covariant action) of an elementary sketch is comprised of:

1. an assignment of a set X_A to each sort X and
2. an assignment of a function $\tau_* : X_A \to Y_A$ for each operation-symbol of the appropriate arity such that:
3. each law is preserved; i.e., for each law as before we have

$$\tau_{n*}\left(\tau_{n-1*}\left(\cdots\tau_{2*}\left(\tau_{1*}\left(x\right)\right)\cdots\right)\right) = \sigma_{m*}\left(\sigma_{m-1*}\left(\cdots\sigma_{2*}\left(\sigma_{1*}\left(x\right)\right)\cdots\right)\right).$$

That is, the covariant action on operation-symbols is faithful in the sense defined above.

The next definition will feature prominently in our later study of type theory, but will also prove immediately useful in studying Example 1.2.2 by forming the sets of a "for-free" model for any elementary sketch.

Definition 1.2.6. Given an elementary (unary) sketch, the **clone** at (Γ, X) is the set $\mathrm{Cn}_{\mathcal{L}}\left(\Gamma, X\right)$ of all the **terms** of sort X assuming a single variable of sort Γ, quotiented by the laws of the sketch.

The fact that a sketch's clones contain *equivalence classes* (with respect to the laws) of its terms will feature prominently in our later study of ideas central to

the goals of this thesis. In particular, clones alone don't allow for any meaningful discussion of computational behavior of terms undergoing reduction; a term's normal form and its various reducible forms are identified in the clone.

It can be shown that the clones of a sketch form (the sets for) a model of a sketch. In particular, it can be shown that the sketch acts covariantly on the set of its clones:

Theorem 1.2.7. Every elementary sketch has a faithful covariant action on its clones $\mathcal{H}_X = \cup_\Gamma \mathrm{Cn}_\mathcal{L}(\Gamma, X)$ by sequencing with the operation symbol. Substitution for the (single) variable in a term gives a faithful contravariant action on $\mathcal{H}^Y = \cup_\Theta \mathrm{Cn}_\mathcal{L}(Y, \Theta)$.

Proof. The actions of $\tau : X \to Y$ on $\mathrm{Cn}_\mathcal{L}(\Gamma, X) \subseteq \mathcal{H}_X$ and $\mathrm{Cn}_\mathcal{L}(Y, \Theta) \subseteq \mathcal{H}^Y$ are given by:

- $\tau_* a_n\left(\cdots a_2\left(a_2\left(\sigma\right)\right)\cdots\right) = \tau\left(a_n\left(\cdots\left(a_2\left(a_1\left(\sigma\right)\right)\right)\right)\right) \in \mathrm{Cn}_\mathcal{L}(\Gamma, Y)$
- $\tau^* \zeta_m\left(\cdots \zeta_2\left(\zeta_1\left(y\right)\right)\right) = \zeta_m\left(\cdots \zeta_2\left(\zeta_1\left(\tau\left(x\right)\right)\right)\right) \in \mathrm{Cn}_\mathcal{L}(X, \Theta)$

where $\sigma : \Gamma, x : X$ and $y : Y$. Covariance of the former is clear. Contravariance of the latter follows by considering the behavior of substitutions in sequence. □

Recalling our sketch of a monoid from Example 1.2.2, the substance of this covariant action morally amounts to saying that the sketch acts on its terms by left multiplication (here "multiplication" is actually just juxtaposition plus some parentheses) which gives the robust version of the handwavy argument we provided above.

Joke 1.2.8. A couple of type theorists walk into a Michelin starred restaurant. The menu reads in blackboard bold letters ``$\mathbb{NO}\ \mathbb{SUBSTITUTIONS}$''. They promptly leave.

The reader might have noted that there is a strong resemblance between the notion of a sketch and that of a category. Indeed, the following result uses the clone action defined above to show that we can associate to every sketch a category and that to every category we can associate a sketch. These assignments moreover induce an isomorphism of categories.

Theorem 1.2.9 (The canonical elementary language)**.** Every elementary sketch $\mathcal{L}$ presents a category $\mathrm{Cn}_\mathcal{L}$ via the for-free action on its clones, and conversely any small category C is presented by some sketch $\mathcal{L}$ in the sense that $C \cong \mathrm{Cn}_\mathcal{L}$. We write $\lceil - \rceil$ for this isomorphism and call the sketch $L(C) = \mathcal{L}$ the *canonical elementary language* of C.

The language is defined as follows:

- The sorts $\lceil X \rceil$ of $L(C)$ are the objects X of C
- The operation symbols $\lceil f \rceil$ are the morphisms f of C and
- The laws are $\lceil \mathrm{id} \rceil(x) = x$ and $\lceil g \rceil\left(\lceil f \rceil\left(x\right)\right) = \lceil g \circ f \rceil\left(x\right)$

and the isomorphism $C \cong \mathrm{Cn}_\mathcal{L}$ is clear.

1.3 The category of contexts and substitutions

We now introduce a category which is special in both the structure it enjoys as well as the central role it will play in the rest of this thesis. This category goes by many names: *syntactic category*, the verbose *category of contexts and substitutions*, and the elusive *classifying category*. We endeavor to explain the meaning behind each of these names over the course of the thesis, but for now we adopt the name most closely describing its presentation.

Definition 1.3.1 (The category of contexts and substitutions)**.** Given a sketch $\mathcal{L}$, the **category of contexts and substitutions**, written $\mathrm{Cn}^{\times}_{\mathcal{L}}$ is presented as follows:

- The objects are the contexts of $\mathcal{L}$, i.e., finite lists of distinct variables and their types.
- The generating morphisms are:
  - Single substitutions or *declarations* $\left[a/x\right] : \Gamma \to [\Gamma, x : X]$ for each term $\Gamma \vdash a : X$. The direction in the signature should be confusing unless you're either already an expert or a total novice to type theory.
  - Single omissions or *drops* $\hat{x} : [\Gamma, x : X] \to \Gamma$ for each variable $x : X$.
- The laws are given by an extended version of the substitution lemma from type theory [Pie02]. When talking about these laws, we will often call them "the extended substitution lemma." The following laws are imposed for each collection of terms a, b and distinct variables x and y such that x does not appear free in a and y appears free in neither a or b:

$$\begin{aligned} \left[a/x\right] ; \hat{x} &= \mathrm{id} \\ \left[a/x\right] ; \left[b/y\right] &= \left[\left[a/x\right]^* b/y\right] ; \left[a/x\right] \\ \left[a/x\right] ; \hat{y} &= \hat{y}; \left[a/x\right] \\ \hat{x}; \hat{y} &= \hat{y}; \hat{x} \\ \left[x/y\right] ; \hat{x}; \left[y/x\right] ; \hat{y} &= \mathrm{id}. \end{aligned}$$

We will briefly speak to the meaning of the laws. The first law says that binding a variable to some term and then forgetting the variable is just the same as doing nothing. The second law says that successive variable declarations commute *up to accounting for the first declaration in the body of the second*. The third law says that *non-overlapping* declarations and drops commute. The fourth law says that pairs of non-overlapping drops commute. The last law is tricky and is easier to explain by passing to the substitution point-of-view. Since the substitution action is contravariant, this requires considering the compositions in reverse order as:

$$\hat{y}^*; \left[y/x\right]^* ; \hat{x}^*; \left[x/y\right]^* = \mathrm{id}^*.$$

Rendered thus, this law means that introducing a free variable y to the context[2], followed by replacing every free occurrence of x with y, followed by re-introducing x as a variable in the context, and then finally replacing every free occurrence of y with x is the same as doing nothing. More concisely at the expense of precision, renaming a free variable in a term and then un-renaming it results in the same term.

We can give the syntactic category a model in the clones by acting by substituting for variables, along the lines of Theorem 1.2.7. The reader unfamiliar with substitution will find relief in Example 1.5.6, where we show how this action works in terms of a concrete (and visual) example from digital logic.

Remark 1.3.2. The morphisms of the category of contexts and substitutions $\mathrm{Cn}_{\mathcal{L}}^{\times}$ for some sketch $\mathcal{L}$ acts contravariantly on the clones $\mathcal{H}^Y = \cup_X \mathrm{Cn}_{\mathcal{L}}(Y, X)$. In the case of single substitution $\left[a/x\right]$, the action is by substituting the term a for the variable x. In the case of weakenings, the action has no operational or syntactic significance.

This category allows us to define a special class of functor. In our case, that class of functor captures what it means to produce a model of an elementary sketch. The proof of this theorem is rather bureaucratic, but its importance is that it teaches us that the canonical elementary language of a category is purpose-built so that its models are precisely set-valued functors out of the category in question.

1.4 Models are essentially functors

Theorem 1.4.1 (The classifying category)**.** Let $\mathcal{L}$ be an elementary sketch and $\mathrm{Cn}_{\mathcal{L}}$ the category it presents. Then the models of $\mathcal{L}$ correspond to functors $\mathrm{Cn}_{\mathcal{L}} \to \mathfrak{Set}$.

Proof. ($\Rightarrow$) Suppose we have an $\mathcal{L}$-model A. Then A is an assignment of a set $\lceil X \rceil_A$ to each sort $\lceil X \rceil$ and an assignment of a function $\lceil r \rceil_A : X_A \to Y_A$ to each operation-symbol $X \vdash \lceil r \rceil (x) : Y$ such that the laws (given by Theorem 1.2.9) of $\mathcal{L}$ are preserved. These assignment form precisely the data of a functor

$$\begin{aligned} F_A : \mathrm{Cn}_{\mathcal{L}} &\to \mathfrak{Set} \\ X &\mapsto \lceil X \rceil_A \\ \left(X \xrightarrow{r} Y\right) &\mapsto \lceil r \rceil_A. \end{aligned}$$

It remains to show functoriality of these assignments which follows from the laws of the canonical elementary language and faithfulness of the model.

($\Leftarrow$) Suppose we have a functor $F_A : \mathrm{Cn}_{\mathcal{L}} \to \mathfrak{Set}$. We will construct a model A of $\mathcal{L}$ from F_A as follows: Recall from Theorem 1.2.9 that the sorts $\lceil X \rceil$ of the sketch $\mathcal{L}$ are precisely the objects X of $\mathrm{Cn}_{\mathcal{L}}$, and the operation symbols $X \vdash \lceil f \rceil : Y$ are the morphisms f of $\mathrm{Cn}_{\mathcal{L}}$. Now,

[2]Note that weakening by y might have no effect if y is already present as a free variable.

1. For each sort $\lceil X \rceil$ we assign $\lceil X \rceil_A = F_A(X)$.

2. For each operation symbol $\lceil f \rceil$ we assign $\lceil f \rceil_A = F_A(f)$.

3. Again by Theorem 1.2.9, the only laws of the sketch are that $\lceil \mathrm{id} \rceil(x) = x$ and $\lceil g \rceil \left(\lceil f \rceil (x) \right) = \lceil g \circ f \rceil (x)$. According to the assignments in the previous two points, the first law says that $F_A(\mathrm{id}_X) = \mathrm{id}_{\lceil X \rceil_A}$, and the second says that $\left(F_A \lceil g \rceil\right) \circ \left(F_A \left(\lceil f \rceil \right) \right) = F_A(g \circ f)$. Both are ensured by functoriality.

□

1.5 Algebraic theories and their algebras

While conjuring up example elementary sketches, the reader will find that the regime of unary operations regularly gets in the way of representing familiar gadgets like rings or type theories. To address this deficiency, we introduce the doctrine of *algebraic theories* which generalizes the doctrine of elementary sketches and allows for multi-input operation symbols. As we work through this upgrade in doctrine, many of the notions (*terms, clones, syntactic category, etc.*) we developed in the simplified world of elementary sketches will come along for the ride without much changed.

Definition 1.5.1 (Algebraic theory)**.** A (finitary many-sorted) **algebraic theory** $\mathcal{L}$ has

1. a collection Σ of base types or **sorts** X

2. an inexhaustible collection of variables $x_i : X$ of each sort;

3. a collection of **operation symbols**, $x_1 : X_1, \ldots, x_k : X_k \vdash r(x_1, \ldots, x_k) : Y$ each having an **arity**, namely a list of input sorts X_i, and an output sort Y; and

4. a collection of **laws**, posed as equalities between different terms (in the sense defined before)

We will write $\mathbb{1}$ for the nullary product of sorts, and moreover when working with a *constant* operation symbol $\mathbb{1} \vdash c() : X$, we will often allow ourselves to simply write c rather than $c()$.

Remark 1.5.2. The syntactic category $\mathrm{Cn}^{\times}_{\mathcal{L}}$ of an algebraic theory is given as before for elementary sketches. Because the syntactic category defined for elementary sketches already has products given by *contexts*, there is nothing more to do.

The next major concept we will introduce generalizes to algebraic theories the notion of *action* or *model* we saw previously for elementary sketches. As expected, the definition will be essentially the same up to taking some products.

Definition 1.5.3 ($\mathcal{L}$-algebra)**.** Given an algebraic theory $\mathcal{L}$ and a category C with finite products (in the sense of the universal property as treated in the chapter on basic category theory) an *$\mathcal{L}$-algebra in C* is comprised of

1. an object A_X of C for each sort X of $\mathcal{L}$, and
2. for each operation symbol $X_1, \ldots, X_k \vdash r : Y$, an assignment of a map $r_A : A_{X_1} \times \cdots \times A_{X_k} \to A_Y$ in C.

such that the assignments respect the laws of $\mathcal{L}$.

We now give some prototypical example theories and algebras. The first such pair we consider is the familiar structure of a ring from abstract algebra, one of whose algebras is the ring of integers under addition and multiplication.

Example 1.5.4 (Algebraic theory of *ring*)**.** We sketch an algebraic theory encoding the familiar structure of a ring from abstract algebra. The presentation should look familiar (when squinting) to anyone with a background in abstract algebra, except that we force the existence of multiplicative and additive identities by requiring any model (to be defined!) of this theory to 1) provide *global elements*, namely morphisms out of the terminal object of the category hosting the model and 2) satisfy the defining equations of the identity.

1. Sorts: The single sort is S. The variable collection for the sort is $\{x, y, z\} \cup \{ s_i \}_{i \in \mathbb{N}}$.
2. Operations:

$$\begin{aligned} \cdot &: S \times S \to S, \\ + &: S \times S \to S, \\ 0 &: \mathbb{1} \to S, \\ 1 &: \mathbb{1} \to S, \\ - &: S \to S. \end{aligned}$$

3. Laws:

$$\begin{aligned} +(x, y) &= +(y, x) \\ +\left(0\,(), x\right) &= x \\ +\left(x, -(x)\right) &= 0\,() \\ \cdot(x, y) &= \cdot(y, x) \\ \cdot\left(1\,(), x\right) &= x \\ \cdot\left(x, +(y, z)\right) &= +\left(\cdot(x, y), \cdot(x, z)\right). \end{aligned}$$

Note that the countably infinite set of variables $\{s\}_{i \in \mathbb{N}}$ is added so that we may have deeply nested expressions, even if we had no need for them when stating the laws

above. In fact, such countable variable sets are *required* when defining an algebraic theory for precisely this reason. That this requirement was not imposed for elementary sketches is because there can only be one variable in a term, even in heavily nested terms, by construction.

Having given some concrete examples, we now allow ourselves another abstract definition: that of an $\mathcal{L}$-*algebra* for an algebraic theory:

Example 1.5.5 (The ring of integers)**.**

1. For the single sort, we set $A_S = \mathbb{Z}$
2. For the operations, we set
   (a) $\cdot_A$ to be the ordinary multiplication of integers
   (b) $+_A = +$ where the second plus is ordinary addition of integers
   (c) 0_A to the function $* \mapsto 0 \in \mathbb{Z}$
   (d) 1_A to the function $* \mapsto 1 \in \mathbb{Z}$
   (e) $-_A$ to the function $x \mapsto -x$ taking an integer to its additive inverse
3. Verifying the rest of the laws is routine after understanding how to verify any one of them, so we demonstrate just one. We show that the 0 selected by the model indeed serves as the left identity of addition in the model.

   Proof.

$$\begin{aligned} +_A \circ \langle 0_A(*), \mathrm{id} \rangle &= (y : \mathbb{Z}) \mapsto 0_A(*) + \mathrm{id}(y) \\ &= (y : \mathbb{Z}) \mapsto 0_{\mathbb{Z}} + y \\ &= (y : \mathbb{Z}) \mapsto y \\ &= \mathrm{id}_{S_A} \end{aligned}$$

   which is the interpretation of the variable x, as required. □

The reader familiar with algebra will observe that this example amounts to verifying that the integers form a ring under the standard multiplication and addition operations we learn in elementary school. In general, the reader will find that all *rings* (in the conventional sense of abstract algebra) are $\mathfrak{Set}$-valued *algebras* of the algebraic theory of *ring* and that conversely every $\mathfrak{Set}$-valued *algebra* of the theory of *ring* defines *a ring*. The reader may also verify that the usual ring-homomorphisms in $\mathfrak{Set}$ are an instance of the generalized notion of $\mathcal{L}$-algebra homomorphism that we introduce following an intervening example theory-algebra pair from computing. Conversely,

This example is meant for readers with a basic background in digital logic. An interesting aspect of this example is that it admits some very practical algebras.

Example 1.5.6 (The algebraic theory of digital logic circuits)**.** We can encode the language of digital logic circuits as an algebraic theory, presented as follows:

1. The single sort is $\mathbb{B}$. We will write $\mathbb{B}^k$ for the k-wide product $\mathbb{B} \times \mathbb{B} \times \cdots \times \mathbb{B}$ for finite k.
2. The variable collection is $\{b_k \mid k \in \mathbb{N}\}$.
3. The operations, for each finite $k \in \mathbb{N}$, are:

$$\begin{aligned} \texttt{AND}_k &: \mathbb{B}^k \to \mathbb{B} \\ \texttt{OR}_k &: \mathbb{B}^k \to \mathbb{B} \\ \texttt{NOT} &: \mathbb{B} \to \mathbb{B} \\ \top &: \mathbb{1} \to \mathbb{B} \\ \bot &: \mathbb{1} \to \mathbb{B} \end{aligned}$$

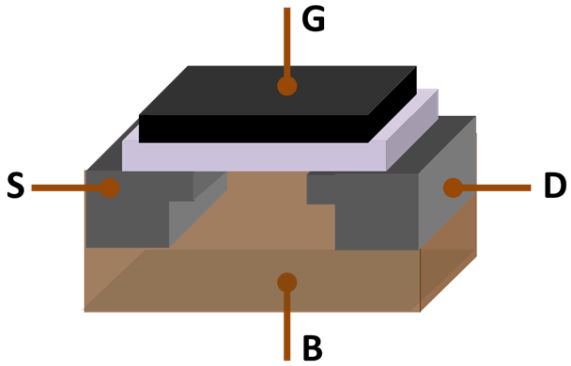
4. The laws, for each finite $k \in \mathbb{N}$, are:

$$\begin{aligned} \texttt{AND}_k\,(b_1, b_2, \ldots, \bot, \ldots, b_k) &= \bot \\ \texttt{OR}_k\,(b_1, b_2, \ldots, \top, \ldots, b_k) &= \top \\ \\ \texttt{AND}_k\,(\top, \ldots, \top) &= \top \\ \texttt{OR}_k\,(\bot, \ldots, \bot) &= \bot \\ \\ \texttt{NOT}\,(\top) &= \bot \\ \texttt{NOT}\,(\bot) &= \top \end{aligned}$$

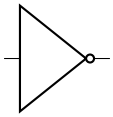
A mundane but useful algebra for this theory is given as follows: We assign the the single sort to be the set $\{0, 1\}$. For the operation symbols, we assign $\top$ to the function $* \mapsto 1$; $\bot$ to function $* \mapsto 0$; $\texttt{AND}_k$ to the function which multiplies all of its inputs; $\texttt{OR}_k$ to be the function which takes the maximum of its inputs; and finally not to be the function taking b to $1 - b$. This algebra provides a convenient way of writing down and quickly reducing boolean circuits according to the familiar symbolic language of algebra.

A more interesting algebra of this theory comes from practical engineering. We can give an algebra of the theory above by implementing the *logic gate* operations NOT, AND, and OR as component circuits built with *metal oxide semiconductor field effect transistors*, also known as *MOSFETs*. You probably own a couple billion of them. The reader more familiar with computer games than MOSFETs might be interested in Nguyen's work which exhibits an algebra for the same theory in *Minecraft* [Ngu]. We do not consider this algebra here, instead focusing on the physical algebra.

Figure 1.1: A diagram of a MOSFET, by Brews ohare - CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=18796795



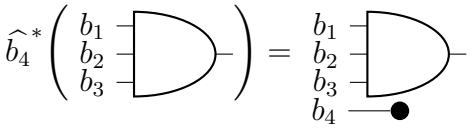
The exact details of how a MOSFET is used to construct such circuits are beyond the scope of this thesis; we refer the interested reader to Ben Eater's videos on the constructions [Eat15]. The particularly interested reader might also consult a standard textbook [BV14] for more on the matter. For our purposes, the upshot of this construction is that we get, for say the NOT operation, a circuit component



which has low voltage on its output when it has high voltage applied to its input, and vice versa. This behavior is precisely the equational law characterizing the NOT operation symbol, and suggests that we interpret the $\top$ operation symbol as a source of high voltage and the $\bot$ operation symbol as a wire with no voltage applied. The AND and OR operations also enjoy physical realizations in term of MOSFETs, as do the finite-input generalizations of all of these operations.

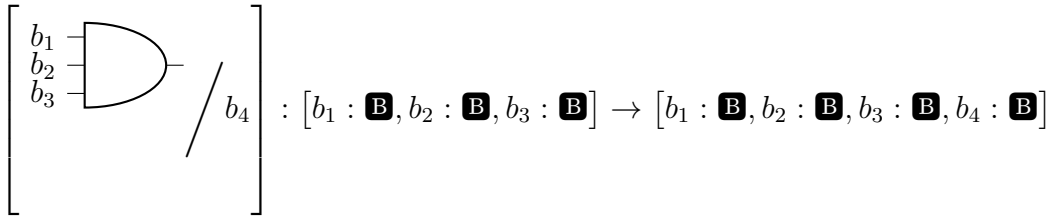
The reader suspicious of abstract nonsense will complain that we have merely dressed up in abstract nonsensical terms the well-known fact that electrical circuits can implement combinational logic. Indeed, the utility of this example lies not in some new exciting insight about digital circuits but rather, as we will soon see, in leveraging our visual intuition about digital circuits to better understand how substitution works, and why substituting for a variable is a contravariant operation.

The circuit interpretation also provides a neat intuition for weakening. Weakening by a variable is drawing a new input wire with the appropriate label, and then not connecting it to anything. As an example, we have the following equality

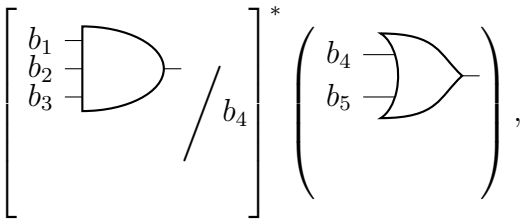


where we represent an unused input as a wire capped off with some sort of insulating material, here displayed as a circle.

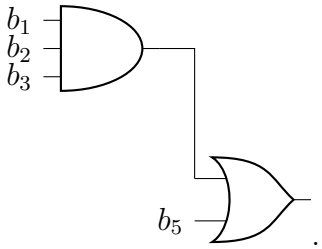
Allowing ourselves to write the operations of the algebra for the operation symbols of the theory for a moment, consider the morphism



in the syntactic category. We can now consider a concrete example of substitution in action by substituting along this morphism. Ignoring the requisite intermediate weakenings, we have



which results in the circuit component



Looking back at the morphism encoding this substitution operation, we should remind ourselves that the b_1, b_2, b_3 in the codomain context were not necessarily *used* in the original circuit; indeed, those variables could be (as in this case) merely unused inputs which were added by a weakening operation. What is essential is that such a weakening *has put the input wires in place* before we slot the 3-wide AND into place.

Definition 1.5.7 ($\mathcal{L}$-algebra homomorphism)**.** A *homomorphism* $\phi : A \to B$ of $\mathcal{L}$-algebras A and B in some category C with finite-products is an assignment to each sort X of a C-morphism $\phi_X : A_X \to B_X$ between the corresponding objects in each algebra such that diagrams of the following form commute:

$$\begin{array}{ccc} A_{X_1} \times \cdots \times A_{X_k} & \xrightarrow{\quad r_A \quad} & A_X \\ \downarrow{\scriptstyle \phi_{X_1} \times \cdots \times \phi_{X_k}} & & \downarrow{\scriptstyle \phi_X} \\ B_{X_1} \times \cdots \times B_{X_k} & \xrightarrow{\quad r_B \quad} & B_X \end{array} .$$

Remark 1.5.8. The following is an observation due to Lawvere in a paper written in his time teaching at Reed College [Law63]. The familiarity of this diagram is no mistake: indeed, by analogy to Theorem 1.4.1, we may understand algebras as product-preserving functors. A mapping between algebras then is a natural transformation, hence the naturality diagram in Definition 1.5.7. We will later make this analogy more concrete in Theorem 1.5.11.

Remark 1.5.9. Perhaps predictably, the C-valued algebras and homomorphisms of an algebraic theory $\mathcal{L}$ form a category, called $\mathrm{Mod}_C(\mathcal{L})$.

Proof. Per Remark 1.5.8, we consider algebras and their homomorphisms as functors and natural transformations respectively. The identity morphisms are the identity natural transformations whose component morphisms are the identities of C. Composition of natural transformations is given by composition of their component morphisms, hence we may out-source the associativity condition to that guaranteed by the categorical structure of C. □

Many things you'd like to prove about a type theory are concerned with the *terms* of the theory at hand. *Normalization* theorems talk about the accessibility (under some reduction relation) of a certain class of terms from any arbitrary open term. *Canonicity* or *closed normalization* theorems talk about the accessibility of a different class of terms from arbitrary *closed* terms. These are but two examples of a broad spectrum of properties one might desire of the terms of a theory. Considering the primacy of term properties in type theory, it is rather strange that the notion of semantics we have built so far makes no commentary on terms besides the action on the clones given in Theorem 1.2.7. Our models so far have only given meaning to the individual *sorts* (types) and individual *operation symbols* (constructors) of the theory considered. In fact, this is enough: our models extend canonically *along the product structure* to contexts and substitutions and thus give meaning to terms.

Definition 1.5.10 (Extending a model to terms)**.** Let A be an $\mathcal{L}$-algebra in a category C. This algebra extends canonically to an interpretation $[\![-]\!]$ of contexts by the following definition recursive in the structure of contexts:

$$[\![\varnothing]\!] = \mathbb{1}_C \tag{1.1}$$

$$[\![\Gamma, x : X]\!] = [\![\Gamma]\!] \times A_X. \tag{1.2}$$

The (overly) careful reader will complain that C doesn't necessarily feature a terminal object, but it turns out that a terminal object is guaranteed[3] by the finite product closure we imposed on C in our definition of algebras. We are good to go.

Recalling more from the definition of an algebra, we know that A gives meaning to each operation symbol $Y_1, \ldots, Y_k \vdash r : Z$ as a morphism $r_A : A_{Y_1} \times \cdots \times A_{Y_k} \to A_Z$ and gives meaning to each constant $c : Z$ by a morphism $\mathbb{1}_C \to A_Z$. We can extend this to arbitrary terms in the context $\Gamma \equiv [\![x_1 : X_1, \ldots, x_n : X_n]\!]$ by the following

[3] I.e., because the terminal object is just the nullary finite product.

recursive definition:

$$\begin{aligned}
[\![x_i]\!] : [\![\Gamma]\!] &\equiv A_{X_1} \times \cdots \times A_{X_n} \xrightarrow{\pi_i} A_{X_i} \\
[\![c]\!] : [\![\Gamma]\!] &\xrightarrow{<_!} \mathbb{1}_C \xrightarrow{c_A} A_Z \\
[\![r\,(u_1, \ldots, u_k)]\!] : [\![\Gamma]\!] &\xrightarrow{\langle [\![u_1]\!], \ldots, [\![u_k]\!] \rangle} A_{Y_1} \times \cdots \times A_{Y_k} \xrightarrow{r_A} A_Z
\end{aligned}$$

where the $[\![u_i]\!]$ are the interpretations of the sub-expressions of the expression in the final line, π_i is the ith projection guaranteed to us by the universal property of products, and $<_!$ is the unique map into the terminal object. The angle bracket notion is used to express the product functor's action on morphisms in C. For clarity, we write out explicitly the composites for the reader:

$$\begin{aligned}
[\![x_i]\!] &\equiv \pi_i \\
[\![c]\!] &\equiv c_A \circ <_! \\
[\![r\,(u_1, \ldots, u_k)]\!] &\equiv r_A \circ \langle [\![u_1]\!], \ldots, [\![u_k]\!] \rangle\,.
\end{aligned}$$

Theorem 1.5.11 (The classifying category of an algebraic theory)**.** Let $\mathcal{L}$ be an algebraic theory. Then $\mathrm{Cn}_{\mathcal{L}}$ has finite products and

1. there exists in $\mathrm{Cn}_{\mathcal{L}}$ an $\mathcal{L}$-algebra which satisfies the following universal property:
2. Let C be another category with finite products and its own an $\mathcal{L}$-algebra. Then the functor $[\![-]\!] : \mathrm{Cn}_{\mathcal{L}} \to C$ preserves finite products and the $\mathcal{L}$-algebra, and is the unique such functor.

Proof. We first show that the syntactic category has finite products. Recall that the objects of the syntactic category are variable contexts $[x : X, y : Y, \ldots]$. For any other context $[t : T, u : U, \ldots]$ we have the product

$$[x : X, y : Y, \ldots] \times [t : T, u : U, \ldots] = [x : X, y : Y, \ldots, t : T, u : U, \ldots]$$

That is, products are given by concatenation of contexts. The projections are given by weakening by all the variables in either context: that is, the second projection is $\Gamma \times \Delta \xrightarrow{\hat{\gamma_1} \circ \cdots \circ \hat{\gamma_k}} \Delta$ where the γ_i are the variables of Γ. We define the first projection similarly. The universal property for products is upheld by the equational laws concerned with weakenings in the category of contexts.

Now come the more interesting promised results:

1. (a) The sorts X of $\mathcal{L}$ are interpreted as single variable contexts $[x : X] \in \mathrm{ob}\, C$ where the variable x is arbitrary.
   (b) The operation symbols $X_1, X_2, \cdots \vdash r : Y$ of $\mathcal{L}$ are interpreted as substitutions $\left[r\,(x_1, x_2, \ldots)\,/y\right] : [x_1 : X_1, x_2 : X_2, \ldots] \to [y : Y]$
2. The functor promised is precisely the one defined by Definition 1.5.10. Preservation of the model and of finite products both follow from the definition of the interpretation by induction on contexts. Uniqueness of the functor is in turn forced by preservation of the model and finite products.

□

Chapter 2

Functorial semantics of the simply typed lambda calculus in cartesian closed categories

"Eeny, meeny, miny, moe"

Alonzo Church (Allegedly, on his choice of λ as the name for his calculus.)

This chapter exploits the heavy machinery developed in the previous chapter to give meaning, in specially structured categories, to the types and terms of the simply typed lambda calculus. Here is how we shall do so: First, we will present various a suite of simple type theories as a series of *algebraic theories* differing only in their equational laws. One of these type theories will be the typical simply typed lambda calculus with the full gamut of equational laws, including β-conversion and η-conversion and α-renaming. Second, we present an interesting perspective on a well known type of categorical structure, namely *cartesian closedness*, by explicitly defining it in terms of the definitional β and η laws from type theory. We will find that this characterization exactly coincides with the one found in typical books on category theory. Finally, we will show that any interpretation of base types of the simply typed lambda calculus gives rise to an interpretation of lambda terms in any *cartesian closed* category with an interpretation of the base types. Unless otherwise noted, what follows is an adaptation of [Tay99, Chapter 4.7]. We begin by presenting the term language and type system for the calculus under consideration.

2.1 The simply typed lambda calculus

We'll work with the simply typed lambda calculus with some collection T of *base types* (say natural numbers, integers, booleans, or some other type of object the reader is interested in). We will not dwell on the details of standard aspects of this development

and instead refer the reader to the standard references [Pie02], [Har16] for the full story.

Definition 2.1.1 (Types)**.** Given a set of base types T, the types $\widetilde{T}$ of the simply typed lambda calculus are generated by the following grammar:

$$\tau ::= \mathbb{1} \mid \tau \to \tau \mid \tau \times \tau \mid \theta$$

where θ is a *base type* drawn from T. Note that the unit type $\mathbb{1}$ is not a base type but rather can be considered a nullary product of types.

Definition 2.1.2 (Terms)**.** The terms of the simply typed lambda calculus (with booleans) are generated by the following grammar:

$$t ::= x \mid () \mid \pi_1\, t \mid \pi_2\, t \mid (t_1, t_2) \mid t_1\, t_2 \mid \lambda x : \tau.\, t$$

where the variables x are drawn from a countably infinite set V.

The typing rules are given as follows:

Definition 2.1.3 (Typing rules)**.**

$$\frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \pi_i t : \tau_i} \qquad \frac{\Gamma \vdash t_i : \tau_i}{(\tau_1, \tau_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau' \to \tau \qquad \Gamma \vdash t_2 : \tau'}{\Gamma \vdash t_1 t_2 : \tau} \qquad \frac{\Gamma, x : \tau' \vdash t : \tau}{\Gamma \vdash \lambda x : \tau'.\, t : \tau' \to \tau}$$

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash () : \mathbb{1}}$$

2.2 Raw cartesian closed structure, beta-eta rules, and cartesian closed structure

The following notion mediates between *having lambda abstraction* and the more structured situation enjoyed by usual presentations of the lambda calculus. The mediating notion, namely *raw cartesian closed structure*, merely requires that one be able to write down lambda abstractions, and that they behave nicely with respect to substitutions that don't interfere with the bound variable. Notably lacking in *raw cartesian closed structure* from the more structured situation referred to above are the *β- and η-laws* which explain how to *compute* with lambda abstraction.

The definition comes from Taylor [Tay99].

Definition 2.2.1 (Raw cartesian closed structure)**.** Let C be a category with finite products together with product projections. We say that C is *raw cartesian closed* or *has raw cartesian closed structure* if we have the following for each pair of objects X, Y of C:

1. An object Y^X
2. A morphism, *application*, $\mathrm{ev}_{X,Y} : Y^X \times X \to Y$ and
3. For each object Γ, a function of hom-sets $\lambda_{\Gamma,X,Y} : C[\Gamma \times X, Y] \to C\left[\Gamma, Y^X\right]$ obeying the naturality law:

$$\lambda_{\Gamma,X,Y}\left(b \circ (u \times \mathrm{id}_X)\right) = \lambda_{\Gamma,X,Y}(b) \circ u$$

for each $u : \Gamma \to \Delta$ and $b : \Delta \times X \to Y$.

Looking closely, the naturality law essentially says that substitutions "not touching" the bound variable commute with lambda abstraction. Informally, this is a good equation to have in mind: $u^*(\lambda x\,.b) = \lambda x.\,(u^* b)$.

We will say that an algebraic theory $\mathcal{L}$ *has raw cartesian closed structure* if its classifying category $\mathrm{Cn}_{\mathcal{L}}^{\times}$ has raw cartesian closed structure.

The following theorem extends Definition 1.5.10 and shows how to construct an interpretation of terms from an algebra in a raw cartesian closed category. The essence is that we extend the assignments of base types and operation symbols along products (as we did before when dealing with ordinary algebraic theories) and along the new exponential objects as well.

Theorem 2.2.2. Let $\mathcal{L}$ be an algebraic theory and C be a category with raw cartesian closed structure and an algebra for $\mathcal{L}$. The algebra for $\mathcal{L}$ extends uniquely along the raw cartesian closed structure to provide an interpretation of terms, which defines a functor $[\![-]\!] : \mathrm{Cn}_{\mathcal{L}} \to C$

Proof. Building on Definition 1.5.10, we define the interpretation by structural recursion:

- The interpretation of base types are given by the algebra for $\mathcal{L}$
- The interpretation of an exponential Δ^Γ is $[\![\Delta]\!]^{[\![\Gamma]\!]}$ where the latter is the exponential entailed by the raw cartesian closed structure.
- Contexts are taken to products as in Definition 1.5.10.
- The variables, operation symbols, and the laws are treated as in Definition 1.5.10.
- The last clause of the raw cartesian closed structure gives the notion of lambda abstraction in C.

□

Definition 2.2.3 (Beta-eta rules)**.** A raw cartesian closed algebraic theory $\mathcal{L}$ satisfies the *β-η rules* if for all $b \in \mathrm{Cn}_{\mathcal{L}}(\Gamma \times X, Y)$ we have the equations

$$\mathrm{ev}_{X,Y} \circ \left(\lambda_{\Gamma,X,Y}(b) \times \mathrm{id}_X\right) = b \qquad (\beta)$$

$$\lambda_{Y^X,X,Y}\left(\mathrm{ev}_{X,Y}\right) = \mathrm{id}_{Y^X}. \qquad (\eta)$$

These equations deserve some commentary. The beta rule says that application of an abstracted body b to the identity gives you back the body you started with: the beta rule forces that application of lambda expressions does nothing more than substitute for the abstracted variable. The eta rule says that taking a function f, applying it to a variable x, then finally abstracting over that variable x is the same operation as doing nothing to f. We will see the more familiar symbolic forms of these rules in the definition of the theory of the usual calculus below.

We now define a notion more well-known outside of computing. Cartesian closed structure endows a category with the ability to take products and function spaces over its objects in a suitable fashion; as can be seen below, the notion of Cartesian product and function space in the category of sets are examples of suitable such constructions. It will turn out that this familiar situation is equivalent to the combination of raw cartesian closed structure and beta-eta rules.

Definition 2.2.4 (Cartesian closed structure)**.** Let C be a category. An *exponential* of objects X and Y is an object Y^X of C together with a morphism $\epsilon : Y^X \times X \to Y$ such that for each object Γ of C and morphism $b : \Gamma \times X \to Y$ there is a unique morphism $\tilde{p} : \Gamma \to Y^X$ called the *exponential transpose* such that $p = \epsilon \circ \left(\tilde{b} \times \mathrm{id}_X\right)$. This is a universal property: exponentials are unique up to unique isomorphism. A category is *cartesian closed* if it has all finite products and all exponentials.

The category $\mathfrak{Set}$ is the prototypical cartesian closed category whose products are the usual cartesian products of sets and whose exponentials are function sets: for sets X and Y, $Y^X = \left\{\text{functions } f \mid \mathrm{dom}\,(f) = X \wedge \mathrm{cod}\,(f) = Y\right\}$. Another good source of example cartesian closed categories is topos theory. All elementary topoi, including categories of presheaves, are cartesian closed [MM92].

It turns out that all of these examples of cartesian closed categories can also be characterized as raw cartesian closed categories that satisfy the beta-eta rules.

Theorem 2.2.5 (Finite-product category + raw CCS + beta-eta $\iff$ cartesian closed)**.** Let C be a category. Then C is cartesian closed if and only if C has finite products and is both raw cartesian closed and satisfies the beta-eta laws.

We will not repeat Taylor's proof [Tay99], but we will say that the proof largely comes down to these two facts: that exponential transposes are unique and that the characterizing property of the transpose is exactly the β rule.

2.3 An algebraic treatment of the lambda calculus

Finally, we will present the lambda calculus as a series of algebraic theories, each of which are identical except for the equations they admit. These theories correspond to instances of the lambda calculus whose respective definitional equalities identify more or fewer lambda terms. We will see that algebras of these theories are essentially functors valued in specially structured categories.

Definition 2.3.1 (Algebraic theory of the lambda calculus)**.** The *α-lambda calculus* is presented as the following algebraic theory called $\mathcal{L}_{\lambda\alpha}$:

1. Sorts: We define the sorts to be the set $\widetilde{T}$ given in Definition 2.1.1.

2. Variables: The variables are defined as the collection $\left\{v : \tau \mid (v, \tau) \in V \times \widetilde{T}\right\}$ where V is the countably infinite set V of untyped variables as in Definition 2.1.2.

3. We define the following operation symbols for each $\tau_1, \tau_2, \tau, \tau' \in \widetilde{T}$:

$$t : \tau_1 * \tau_2 \vdash \pi_1(t) : \tau_1$$
$$t : \tau_1 * \tau_2 \vdash \pi_2(t) : \tau_2$$
$$[t_1 : \tau_1, t_2 : \tau_2] \vdash (t_1, t_2) : \tau_1 * \tau_2$$

$$\left[t : \tau' \to \tau, t' : \tau'\right] \vdash t\, t' : \tau$$

$$\vdash () : \mathbb{1}.$$

   The reader will note that we have not yet added an operation symbol for lambda abstraction. This requires careful consideration, because lambda abstraction is not really an operation symbol but rather a family of operation symbols defined mutually with the others. For each term $\Gamma, x : \tau' \vdash t : \tau$, we add a new operation symbol

$$\Gamma \vdash (\lambda x.\, t)\left(\overrightarrow{\gamma}\right) : \tau$$

   where $\overrightarrow{\gamma}$ is the list of variables comprising Γ.

   Finally, the complete collection of operation symbols is defined as the least set closed under taking abstractions in this way and containing the operation symbols already mentioned above.

4. The only equation is the α-rule which says that terms can be identified up to renamings of their bound variables:

$$\lambda(x : X).\, t = \lambda\left(x' : X\right).\ \left[x'/x\right]^* t. \qquad (\alpha)$$

   Note that we do not include the $\beta\eta$-rules in this version.

With this theory in hand, we first show that it is raw cartesian closed (i.e. that it supports a reasonably coherent notion of lambda abstraction.) After that, we will add to the theory the β and η laws. Finally, we will use the preceding theorems to argue *from β and η* to show that the resulting theory of the typical lambda calculus is cartesian closed.

Theorem 2.3.2 (α-lambda calculus is raw cartesian closed)**.** The theory $\mathcal{L}_{\lambda\alpha}$ is raw cartesian closed

Proof. In the following we write $[X \Rightarrow Y]$ for the exponential Y^X, for the sake of legibility. The exponential

$$\left[x_1 : X_1, x_2 : X_2 \ldots, x_j : X_j\right] \Rightarrow [y_1 : Y_1, y_2 : Y_2, \ldots, y_k : Y_k]$$

is the context $[f_1 : F_1, f_2 : F_2, \ldots, f_k : F_k]$ where the f_i are new variables and

$$F_i = X_1 \Rightarrow \Big(X_2 \Rightarrow \cdots \Rightarrow \big(X_j \Rightarrow Y_i\big) \cdots \Big)$$

Finally, we define $\mathrm{ev}_{\overrightarrow{X},\overrightarrow{Y}}$ as

$$\left[\Big(\cdots \big(f_1 \, (x_1) \, x_2\big) \cdots x_j\Big) \, / y_1\right] \circ \cdots \circ \left[\Big(\cdots \big(f_k \, (x_1) \, x_2\big) \cdots x_j\Big) \, / y_k\right]$$

and $\lambda_{\Gamma,\overrightarrow{X},\overrightarrow{Y}} \left[\overrightarrow{p} / \overrightarrow{y}\right]$ as

$$\left[\Big(\lambda x_1. \, \big(\lambda x_2. \ldots (\lambda x_k. \, p_1) \ldots\big)\Big) \, / f_1\right] \circ \cdots \circ \left[\Big(\lambda x_1. \, \big(\lambda x_2. \ldots (\lambda x_k. \, p_k) \ldots\big)\Big) \, / f_k\right].$$

To clarify the notation, we consider a small example of an exponential object: the exponential

$$[x : X, y : Y] \Rightarrow [z : Z, w : W]$$

is the context

$$\Big[f_z : \big(X \Rightarrow (Y \Rightarrow Z)\big), f_w : \big(X \Rightarrow (Y \Rightarrow W)\big)\Big].$$

Naturality with respect to weakening and single substitution follows from the fact that substitution for (or weakening by) a variable x commutes with abstraction of a distinct variable y: for any terms p and c, we have $\left[c/y\right]^* (\lambda x. \, p) \equiv_\alpha \lambda x. \, \Big(\left[c/y\right]^* p\Big)$. The same situation holds for weakening. The full naturality law follows from this and the fact that single substitution and weakening are the generating morphisms of the syntactic category. □

Definition 2.3.3 (Lambda calculus)**.** The lambda calculus, or the $\alpha\beta\eta$-lambda calculus is defined as the algebraic theory of the α-lambda calculus enriched with the following additional equational laws:

$$\big(\lambda \, (x : X) \, . \, t\big) \; t' = \left[t'/x\right]^* t \qquad (\beta)$$

$$f = \big(\lambda \, (x : X) \, . \, f \, x\big) \, . \qquad (\eta)$$

We write $\mathcal{L}_{\lambda\alpha\beta\eta}$ or simply λ for the new theory, and Cn_λ for its classifying category.

Definition 2.3.4 (Lambda algebra)**.** By a *lambda algebra*, we mean an algebra for the theory $\mathcal{L}_{\lambda\alpha\beta\eta}$.

The following results quickly follow the definition of the theory of the lambda calculus and the work we did in Chapter 1.

Theorem 2.3.5. The lambda calculus is a raw cartesian closed algebraic theory with beta-eta.

Theorem 2.3.6. The syntactic category Cn_λ of the theory $\mathcal{L}_{\lambda\alpha\beta\eta}$ is cartesian closed and has a lambda algebra satisfying the following universal property: if C is a cartesian closed category with a lambda algebra then there is a unique functor $[\![-]\!] : \mathrm{Cn}_\lambda \to C$ that preserves the cartesian closed structure and the lambda algebra.

Proof. Cartesian closure follows immediately from the previous result and Theorem 2.2.5. The rest is given essentially as in Theorem 1.5.11, except that we must handle the new exponentials with more care. Theorem 2.2.2 provides the requisite care and extends the functor defined in Definition 1.5.10 to provide one which preserves exponentials as required. □

This final result will prove important in Chapter 3.

Chapter 3

Normalization by gluing

> You, you were like glue
> holding each of us together
> I slept through July
> while you made lines in the heather
>
> Fleet Foxes, "Lorelai"

The next two chapters will consider the normalization problem for the simply typed lambda calculus using the technology we've developed so far. We begin by reviewing the notion of normalization most likely to be familiar to the reader. Normalization[1] comes in both weak and strong flavors, but both are properties of a *reduction* relation, say, $- \to =$ defined over the terms of a theory. By a *normal form* we mean a term t for which there is no other t' such that $t \to t'$. Writing $- \to^* =$ for the least transitive, reflexive closure of this relation, we say the reduction system enjoys *weak normalization* if for any well-typed term e, there exists a normal form n such that $e \to^* n$. A reduction system enjoying *strong normalization* is one for which *every* reduction sequence ends in a normal term. These are *operational* characterization of normalization; we will see below *equational* characterizations of normalization that better line up with what we have done so far.

Theorems *about theories* (such as a programming language or type system) are called *metatheorems*. Normalization theorems, then, are *metatheorems*. Type safety[2] is another class of metatheorem which is much more famous than normalization, perhaps because some of its variants enjoy straightforward proofs. One variant of this result called *syntactic type safety* defines a set of chosen *good terms* called values and defines a *stuck term* to be a term which is both irreducible and *not* a value. The syntactic type safety theorem says that well-typed terms $\Gamma \vdash t : \tau$ never reduce to a stuck term. This variant of type safety enjoys a straightforward proof by induction over typing derivations using a trick called *progress and preservation*. See Chapters 8 and 9 of Pierce's textbook [Pie02] for the details.

[1] As we will note below, we are talking about an *operational* characterization of normalization here.

[2] This result is often sloganized as *well-typed (closed) terms don't go wrong*.

Normalization theorems on the other hand tend to resist standard techniques like straightforward induction over typing derivations. A famous example is *closed normalization* which says that all closed terms have a normal form. When easy induction fails, metatheoreticians resort to *occult* techniques like the method of logical relations. The method of logical relations is, broadly speaking, opaque to all but its most experienced practitioners. In short, a logical relation is a relation of terms indexed by the types of the theory whose inclusion criteria in some vague sense mirror the type structure of the theory. Such logical relations are easy to work with in simple settings like that of the closed normalization problem for the simply-typed lambda calculus, see Pierce's textbook [Pie02]. However, logical relations are well known to become fiendishly complicated when working with theories whose type structure is more complicated. Examples of such complexity include the addition of type system features associated with computational effects, such as *recursive types* (the effect being non-termination) or *mutable reference types* (the effect being change of *state*). Another variant of type safety presents a great example of the problems associated with using logical relations in the presence of these features.

Type safety has another characterization called *semantic type safety*, as espoused by Milner in his paper introducing the concept of type safety [Mil78]. Proving semantic type safety involves building a $\mathfrak{Set}$-valued model A of the type theory in question where the sets $[\![\tau]\!]_A$ of the model are comprised of terms satisfying a certain behavioral requirement. With this done, a *semantic typing* relation $\Gamma \Vdash t : \tau$ is defined by $t \in [\![\tau]\!]_A$. Finally, one proves the so-called *fundamental* theorem which says that all *syntactically* well-typed terms $\Gamma \vdash t : \tau$ are also *semantically* well-typed, so that all *syntactically* well-typed terms satisfy the behaviorally properties for their type as specified in the model. This type of result is unlike *syntactic* type safety in that it does not admit easy proofs by induction on typing derivations, and rather are typically proven with logical relations. This is no problem for well-behaved type systems like the simply typed lambda calculus, but for type systems with features like *mutable reference types* even (ordinary) logical relations don't suffice and more advanced proof techniques are required. The additional heavy machinery called for is the method of *step-indexed logical relations*, a form of *Kripke logical relations* which I describe below and detail in this chapter. Ahmed, for example, employed *step-indexed logical relations* to prove for the first time the semantic type safety of a version of System F endowed with mutable reference types in her doctoral thesis [Ahm04].

It is not just realistic programming languages whose metatheory demands the use of such heavy machinery, however. The very nature of certain metatheoretical problems preclude the use of ordinary logical relations. One such problem is the *open normalization* problem for the simply typed lambda calculus. Open normalization generalizes the closed normalization problem to terms with free variables: we require that every well-typed term $\Gamma \vdash t : \tau$ has a normal form, in the same sense as above. Such problems again demand that the logical relation be indexed not just by types, but also by the elements of some poset [Harb]. That poset might be, as in the case of the open normalization problem, the poset of contexts Γ ordered by weakenings $\hat{x}$. Such a relation indexed by a poset is called a Kripke relation if it additionally satisfies a certain monotonicity condition. In the example of the previous sentence, these

conditions amount to saying that a term which has a normal form when considered as a term under the context Γ should also have a normal form when considered as a term under any extended context $\Gamma, x', \ldots, z'$. In the case of *step-indexed logical relations*, the indexing poset is $\mathbb{N}$ where the indices represent *remaining reduction steps* and the partial order $m \leq_{\mathtt{step}} n$ is defined e as $n \leq m$. That is not a typo. The monotonicity condition, then, says that a term is in the relation when allowed k steps of reduction is also in the relation when allowed *fewer* steps of reduction [BZ07]. Don't read too much into this; we will be working with something closer to the poset of contexts ordered by weakenings mentioned before.

This chapter will consider the open normalization problem for the simply typed lambda calculus from a slightly different perspective. First of all, the problem we will tackle is not exactly the one I suggested at the beginning of this introduction. Instead of an *operational* normalization theorem, we seek to prove an *equational* normalization theorem, in keeping with the perspective we developed in the previous two chapters. The *equational* version of the normalization problem asks for a given well-typed term $\Gamma \vdash t : \tau$ whether there exists a normal form N such that $[\![t]\!] = [\![N]\!]$, i.e., the two are equal as morphisms in the syntactic category. In fact, our solution will offer even more than that, giving a *function* which computes the corresponding normal form for a given term. Our methods are also substantially different from the usual approach. Building on our work in the previous two chapters, we will develop and make use of the category-theoretic *Artin gluing* construction to tackle this problem. Briefly, the gluing construction allows us to define a category of *proof relevant Kripke logical predicates*, meaning that the objects of the category are Kripke predicates (in the sense of Kripke logical relations mentioned above) with a twist: just as a category is a proof-relevant poset (i.e., morphisms $a \to b$ are witnesses for the proposition $a \leq b$) a proof relevant Kripke logical predicate is one where any inhabitant of the predicate can have multiple *witnesses* of its membership. The predicates are *logical* insofar as the category is *cartesian closed*, which the reader will recognize as saying that it in some sense mirrors the type structure of the lambda calculus.

Where the method of Kripke logical relations is ad-hoc and mysterious, gluing regularizes the thought involved and delivers more conceptual proofs and theory. The gluing construction is perhaps overshadowed in fame by one of its instances: The *scone* or *Sierpinski cone* is well-known outside of type theoretic circles for its use in proving results similar to closed normalization. Because we are interested in open normalization, the scone construction is not suited to the problem at hand. The reader interested in the broader context is encouraged to read on in Section 22 of Part II of Lambek and Scott's *Introduction to higher order categorical logic* [LS89]. Our own case will involve an instance of the gluing construction which, in some sense, glues the syntax of open terms to its meaning in the semantics (in the category Cn_λ). The gluing construction is set up exactly so that morphisms in the category are pairs of *syntactic transformations* and *semantic transformations* (namely substitutions) which in some sense commute with the semantic interpretation of terms in Cn_λ. Our normalization function then will be delivered from a composite of these morphisms, so that the semantic equivalence of the input term and the computed normal form is essentially automatic once we have verified that the maps involved in the composite

are genuine morphisms of the gluing category. In this sense, the gluing category we work with has been set up not just to mirror the type structure of the simply-typed lambda calculus, but to also mirror the desired result.

This chapter will develop the gluing construction itself and define our particular gluing category. There is a lot of technology to build up, and many of the constructions we'll see are non-trivial. *Math is hard, but we'll get through it together!* The subsequent chapter will leverage the technology of this chapter to deliver the long-promised normalization function.

3.1 The comma construction and friends

This section introduces a general construction that will, among other things, allow us to define the gluing construction from which this chapter gets its name. The loose idea of the comma construction is to glob some extra data to the objects of a category in a way that supports a well-defined notion of an object-with-extra-data-morphism which preserves the attached data. This intuition perhaps underlies the name for the gluing construction. Our specific use of the gluing construction will involve gluing presheaves of syntactic terms (as you would write them down on paper) together with presheaves of semantic terms (which are *substitutions*.) That is, the extra data we glob on is the semantics of terms, and the notion of extra-data-preserving morphism is a pair of a syntactic transformation and a substitution which commute with the semantic interpretations of the syntax.

Definition 3.1.1 (Comma category)**.** Let E, D, C be categories and

$$F : D \to C \leftarrow E : G$$

be a pair of functors sharing their codomain C. The comma category $F \downarrow G$ has as its

1. Objects: triples $\left(d : D, Fd \xrightarrow{f} Ge, e : E\right)$
2. Morphisms: $(h, k) : (d, f, e) \to (d', f', e')$ are pairs of D and E-morphisms $h : d \to d', k : e \to e'$ making the following diagram commute:

$$\begin{array}{ccc} Fd & \xrightarrow{Fh} & Fd' \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f'} \\ Ge & \xrightarrow[Gk]{} & Ge' \end{array}$$

Remark 3.1.2 (The gluing construction)**.** When in the above we have $D = C$ and $F = \mathrm{id}_C$, we will write $C \downarrow G$ for the comma category and we will say that we have *glued C to E along G*.

3.2 An easy instance of the comma construction: the category of renamings

As a warm-up example, we present the *category of renamings* which can be nicely expressed as an instance of the comma construction. In elementary terms, the category has the same objects as the syntactic category (i.e., contexts) but the morphisms are a restricted class of substitutions, which are only allowed to substitute for a variable *by a variable of the same type* or weaken by a variable. Thus the category of renamings is something like a less proof-relevant version of the category of contexts and substitutions in the following sense: if there exists a substitution relating two contexts in the syntactic category, then there exists a renaming relating the same contexts. The upshot is that renamings allow us to track changes in context *due to a substitution* without actually working with substitutions directly. Working over Cn_λ is problematic because substitutions (and thus the terms they represent) are identified up to computational equality, whereas we would like to distinguish between terms at differing stages of reduction (e.g. distinguishing between a term and its normal form.) We will understand this problem in greater detail in Remark 4.1.12. Concisely, the category of renamings allows us to define the presheaves we need while not losing track of how contexts evolve by substitution[3]. We present the category abstractly as a comma category and at the same time spell out concretely its morphisms in terms of the substitutions of the syntactic category:

Definition 3.2.1 (Category of renamings)**.** Recall from Chapter 2 that V is some countably infinite set of variables and $\widetilde{T}$ is the set of all types in the simply typed lambda calculus generated from a set of base types T. Let $\mathbb{F}$ be the category whose objects are finite subsets of V and whose morphisms are all functions between the underlying sets. Let $\mathbb{T} : \mathbb{1}_{\mathfrak{Cat}} \to \mathfrak{Set}$ be the functor sending the single object of the terminal category to the set $\widetilde{T}$. Let $U : \mathbb{F} \to \mathfrak{Set}$ be the forgetful functor taking finite variable sets to their underlying sets and functions to functions. Now by the *category of renamings* we mean the *opposite category* of the comma category $U \downarrow \mathbb{T}$ whose

- Objects are *pairs* $\left(V : \mathbb{F}, \Gamma : V \to \widetilde{T}\right)$, i.e., finite variable list together with an assignments of types to a each variable. Note that we elide the final entry of the triple, since it is always the object $* : \mathbb{1}_{\mathfrak{Cat}}$ and is therefore uninformative. These objects are precisely the contexts of the familiar classifying category.

- Morphisms of contexts $(V, \Gamma) \to (V', \Gamma')$ are functions $\rho : V \to V'$ making the

[3]The reader might complain now, as I did, that if we only care about tracking how contexts evolve by substitutions then we should simply work with the category of context *weakenings*. The problem here is that the category of weakenings lacks finite products, which precludes the use of a trick (which we will soon see *does* work in the case of the category of renamings) for explicating the binding structure of its category of presheaves. Thank you to Carlo Angiuli for explaining this to me!

following diagram commute:

$$\begin{array}{ccc} V & \xrightarrow{\rho} & V' \\ {\scriptstyle\Gamma}\downarrow & & \downarrow{\scriptstyle\Gamma'} \\ \widetilde{T} & \xrightarrow[\mathrm{id}_{\widetilde{T}}]{} & \widetilde{T} \end{array}.$$

The reason the morphisms are single functions and not pairs is that the second function would not communicate any data, it would simply be taken by $\mathbb{T}$ to the identity no matter what it is. We now unpack what this means. The functions ρ are *type-preserving renamings*: for each variable $x \in V$, we have that $\Gamma'(\rho x) = \Gamma(x)$. In particular, this restricts ρ to the following classes of morphisms in the classifying category: substitutions of variables for variables of the same type, context extension with new variables, and all compositions of these. The equations are given by the substitution lemma, as in the syntactic category.

We write $\mathrm{Ren}_\lambda = (U \downarrow \mathbb{T})^{\mathrm{op}}$ for the category of renamings.

Remark 3.2.2. We write for the functor $\iota : \mathrm{Ren}_\lambda \to \mathrm{Cn}_\lambda$ which takes contexts to contexts and casts renamings as change-of-variable (or weakening) substitutions. When going between Ren_λ and Cn_λ, we will take the liberty to implicitly insert this coercion as needed without further warning. It turns out that this functor is *faithful* and thus witnesses Ren_λ as a *subcategory* of the syntactic category.

3.3 Variable-arity Kripke relations

In order to motivate the next example of a comma construction, we will define *variable-arity Kripke relations*. Variable-arity Kripke relations are, loosely speaking, families of relations parameterized by the objects of some *category* (c.f. the Kripke logical relations mentioned in the introduction, which vary over the objects of a *poset*) for which inclusion in the relations respects, in a sense to be defined, the morphisms of that category.

Definition 3.3.1 (C-Kripke relation)**.** Let C and S be categories. For a functor $\varsigma : C \to S$, a *C-Kripke relation* R of *arity* ς over an object A of S is a family of sets $\left\{R(c) \subseteq S(\varsigma(c), A)\right\}_{c \in C}$ satisfying the following *monotonicity* condition:

For each morphism $\rho : c' \to c$ in C and every $a : \varsigma(c) \to A$ in $R(c)$, the map $a \circ \varsigma(\rho) : \varsigma(c') \to A$ is in $R(c')$.

In order to disentangle this admittedly abstract definition, we specialize to the case of Ren_λ-Kripke relations which are perhaps more graspable.

Example 3.3.2 (A class of Ren_λ-Kripke relations over Cn_λ)**.** Consider the inclusion functor $\iota : \mathrm{Ren}_\lambda \to \mathrm{Cn}_\lambda$. Following the definition above, we can define a Ren_λ-Kripke relation R of *arity* ι over any type[4] τ of Cn_λ by producing a family of sets

[4]Remember that a type is in particular a singleton context.

$\left\{R(\Gamma) \subseteq \mathrm{Cn}_\lambda[\Gamma, \tau]\right\}_{\Gamma \in \mathrm{Ren}_\lambda}$, namely a selection of terms of type τ in each context Γ, along with a proof of the monotonicity condition: for each *renaming* $\rho : \Gamma' \to \Gamma$ in Ren_λ and every *term* $t : \Gamma \to \tau$ in $R(\Gamma)$, the map $t \circ \iota(\rho) : \Gamma' \to \tau$ is in $R(\Gamma')$. This example demonstrates that we can think of Kripke relations as families of unary relations for which inclusion in any of the relations in the family, say $R(\Gamma)$, those unary relations of terms which can be *lifted* along a renaming to yield a proof of inclusion in the relation at any other context Γ' enjoying a renaming into Γ.

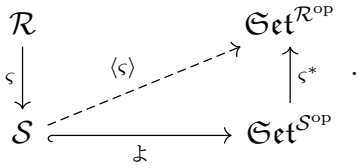
Even with a more concrete example, it can be hard to understand what the utility of a construction like this is without knowing how to map between such C-Kripke relations of a given arity. In fact, we will not say explicitly what a C-Kripke morphism is at all. Instead we turn directly to the work of defining the *gluing category* which turns out to generalize C-Kripke relations to *proof relevant Kripke logical predicates*. Indeed, there is a a category of C-Kripke relations whose objects are defined as above; that category can be found to be a full subcategory of the gluing category which we introduce next. We refer the reader to [Fio02] for more commentary on this result and for a more detailed treatment of Kripke relations in general.

3.4 A harder comma category: the gluing category; or, Artin's monster

With the category of renamings in hand, we can now define *by the comma construction* the gluing category. To define it, we first need to define a functor along which to take the comma. In a loose sense that will be explained in due course, this functor defines Ren_λ-presheaves of *open* (semantic) terms.

The relative hom functor

Every functor $\varsigma : \mathcal{R} \to \mathcal{S}$ induces the following situation:



That is, we get a functor

$$\langle\varsigma\rangle : \mathcal{S} \to \mathfrak{Set}^{\mathcal{R}^{\mathrm{op}}}$$

by adjusting the Yoneda embedding into the category of $\mathcal{S}$-presheaves. Specializing to the case of the inclusion $\iota : \mathrm{Ren}_\lambda \to \mathrm{Cn}_\lambda$ of Remark 3.2.2 gives us a functor

$$\begin{aligned} \mathfrak{Tm} : \mathrm{Cn}_\lambda &\to \widehat{\mathrm{Ren}_\lambda} \\ \Delta &\mapsto \mathrm{Cn}_\lambda\left(\iota(-), \Delta\right) \end{aligned}$$

where $\widehat{\mathrm{Ren}_\lambda}$ is the category $\mathfrak{Set}^{\mathrm{Ren}_\lambda^{\mathrm{op}}}$. This functor can be construed as taking a syntactic context to a presheaf of open terms. This is a confusing idea at first, and it helps to consider the simple cases to understand it. Consider any singleton context $\tau : \mathrm{Cn}_\lambda$. $\mathfrak{Tm}$ takes τ to the presheaf $\mathrm{Cn}_\lambda(-, \tau)$ which takes any renaming context Γ to $\mathrm{Cn}_\lambda(\Gamma, \tau)$. Recalling the definition of the syntactic category and its generating morphisms, the latter set comprises the terms of type τ closed under Γ, represented as single substitutions $\left[t/x\right] : \Gamma \to \tau$. Generalizing to multivariable target contexts Δ gives presheaves of *lists* of open terms of the types in Δ.

It turns out that the category of Ren_λ presheaves shares the cartesian closed structure of Cn_λ and that $\mathfrak{Tm}$ preserves this structure.

Remark 3.4.1. Any category of presheaves, including $\widehat{\mathrm{Ren}_\lambda}$, is cartesian closed. Furthermore, the relative hom functor is a *cartesian closed functor*, meaning that for contexts Γ and Δ we have the following isomorphisms of presheaves:

1. $\mathfrak{Tm}(\Gamma) \times \mathfrak{Tm}(\Delta) \cong \mathfrak{Tm}(\Gamma \times \Delta)$
2. $\mathfrak{Tm}(\Gamma)^{\mathfrak{Tm}(\Delta)} \cong \mathfrak{Tm}\left(\Gamma^\Delta\right)$

We write

$$\mathfrak{Tm}(\Gamma) \times \mathfrak{Tm}(\Delta) \xrightarrow{i_\pi} \mathfrak{Tm}(\Gamma \times \Delta)$$

for the first isomorphism and

$$\mathfrak{Tm}(\Gamma)^{\mathfrak{Tm}(\Delta)} \xrightarrow{i_e} \mathfrak{Tm}\left(\Gamma^\Delta\right)$$

for the second.

The reader should consult Chapter 8 of Awodey's *Category Theory* [Awo10] for more information about these results.

Gluing syntax to semantics along the relative hom functor

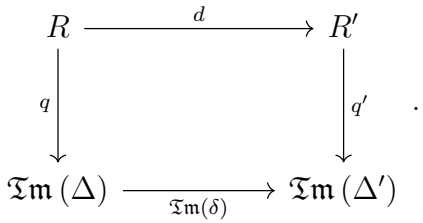
We at last define the *gluing category* as the comma category of the category of renamings taken along the relative hom functor $\mathfrak{Tm}$ just defined.

Definition 3.4.2 (The gluing category)**.** The gluing category Gl_λ is defined as the comma category $\widehat{\mathrm{Ren}_\lambda} \downarrow \mathfrak{Tm}$. Explicitly, its objects are triples

$$\left(R : \widehat{\mathrm{Ren}_\lambda}, q : R \Rightarrow \mathfrak{Tm}(\Delta), \Delta : \mathrm{Cn}_\lambda\right).$$

The objects of the gluing category are *proof relevant Kripke logical predicates*, in a sense that will be more clear after a digression to follow. Following the definition of the comma construction, the morphisms $(R, q, \Delta) \to (R', q', \Delta')$ are pairs $\left(d : R \to R', \delta : \mathrm{Cn}_\lambda(\Delta', \Delta)\right)$ of a $\widehat{\mathrm{Ren}_\lambda}$ natural transformation and a substitution

making the following diagram commute:



We will refer to objects and morphisms in the gluing category as *glued objects* and *glued morphisms* respectively. We do not have a great name for the maps $q : R \Rightarrow \mathfrak{Tm}(\Delta)$ that form part of the data of a glued object. In practice, these maps will often be the semantic interpretation of syntactic terms in Cn_λ, so for glued objects in general we will call q the *interpretation* even if q is not necessarily the usual semantic interpretation.

We can get a foothold understanding of the objects of the gluing category by considering them as a presheaf together with, for each Γ, an *interpretation* q_Γ of each element in $R(\Gamma)$ as a semantic term in $\mathfrak{Tm}(\Delta)(\Gamma) = \mathrm{Cn}_\lambda(\Gamma, \Delta)$ such that the interpretation commutes with renamings in a way that will be further explained in just a bit.

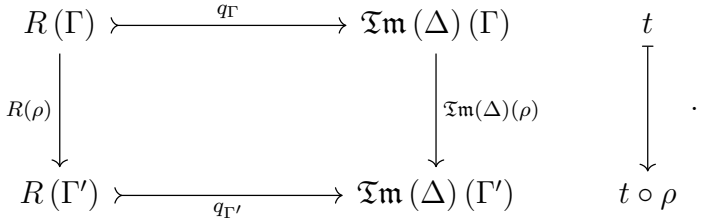
Holding off our desire to further understand the objects, we can already understand the morphisms. A morphism (d, δ) is a natural transformation of Ren_λ-presheaves glued together with a substitution, such that performing on R the transformation d and then looking at the *semantics* of the resultant presheaf R' by interpreting along q' is the same as interpreting the semantics of the original presheaf R along q and then performing a semantic transformation by substituting with δ. This intuition will become more clear when we define glued objects over *presheaves of syntax* for which the interpretation $q : R \Rightarrow \mathfrak{Tm}(\Delta)$ is the actual *interpretation of syntax* in the classifying category.

We can now better understand the objects by understanding how the gluing category subsumes the notion of variable-arity Kripke relations.

The subcategory of (ordinary) variable-arity Kripke relations

We can recover ordinary, proof-*irrelevant* variable-arity Kripke relations as a subcategory of the gluing category. In particular, ordinary variable-arity Kripke relations arise as those objects $\left(D : \widehat{\mathrm{Ren}_\lambda}, q : D \Longrightarrow \mathfrak{Tm}(\Delta), \Delta : \mathrm{Cn}_\lambda\right)$ of the gluing category whose interpretation map q is a component-wise monomorphism; i.e., for each $\Gamma \in \mathrm{Ren}_\lambda$, we have that $q_\Gamma : D(\Gamma) \rightarrowtail \mathfrak{Tm}(\Delta)(\Gamma)$ is a monomorphism. Here's why: recalling the definition of Ren_λ-Kripke relations over $\tau \in \mathrm{Cn}_\lambda$, we need to find in these data a Ren_λ-indexed family $\{R_\Gamma\}_{\Gamma \in \mathrm{Ren}_\lambda}$ of sets of Cn_λ-morphisms into τ subject to the following monotonicity condition: for any renaming $\rho : \Gamma' \to \Gamma$, if $t : \Gamma \to \tau$ is in R_Γ, then we have that $t \circ \rho : \Gamma' \to \tau$ is in $R_{\Gamma'}$. Looking again to our proposed Kripke predicate objects of the gluing category, we can see that the presheaf R together with the natural mono q define for each Γ a subset of the substitutions $\Gamma \to \tau$.

It is tempting to dismiss the naturality of q as bureaucracy, but naturality turns out to be essential to recovering the monotonicity condition. To see this, we need to look at the naturality diagram of q:



As mentioned before, because the components q_Γ of the interpretation are monos we may identify the component morphisms with their images. We write $|q_\Gamma| \subseteq \mathfrak{Tm}(\Delta)(\Gamma)$ for the image, for each Γ. Now what this diagram says is that for any $q_\Gamma\left(r \in R(\Gamma)\right) = t \in |q_\Gamma|$, we have that $t \circ \rho = q_{\Gamma'}\left(R(\rho)(r)\right) \in |q_{\Gamma'}|$. This is precisely the monotonicity condition for Ren_λ-Kripke relations over Δ: containment in each predicate is functorial with respect to renaming *up to renaming*.

The perspective gained above allows for a more conceptual understanding of the objects in the gluing category: the presheaves R define context-indexed families of *witnesses* to the inclusion of terms in the predicate, and each q_Γ is a quotient map *into its image* that forgets the difference between distinct witnesses $w, w' \in R(\Gamma)$ and whose image just records those terms $t \in \mathfrak{Tm}(\Delta)(\Gamma)$ taken to be in the predicate; insofar as q is a mono, these objects are exactly variable-arity Kripke relations over the context Δ.

Chapter 4

Obtaining a normalization function

> I stood on wondering which way to go
> I lit a cigarette on a parking meter
> And walked on down the road, it was a normal day
>
> Bob Dylan

4.1 Presheaves of neutral and normal syntax

In this section, we will enlist a motley crew of *presheaves of syntax* which more-or-less represent certain classes of terms in the lambda calculus. These presheaves are defined over Ren_λ rather than Cn_λ for reasons that will be detailed in Remark 4.1.12. We will ultimately define lambda algebras over these presheaves, for which we require some understanding of how to represent variables in $\widehat{\mathrm{Ren}_\lambda}$ and how exponentiation in $\widehat{\mathrm{Ren}_\lambda}$ corresponds to lambda abstraction.

4.1.1 Variables and binding in Renhat

Definition 4.1.1 (Variable presheaf)**.** We can define a presheaf of *typed variables* in $\widehat{\mathrm{Ren}_\lambda}$ with the Yoneda embedding on Ren_λ:

$$\mathfrak{V}_\tau = \text{よ}\tau = \mathrm{Ren}_\lambda\left(-, \tau\right).$$

With that definition, we have $\mathfrak{V}_\tau\left(\Gamma\right) = \mathrm{Ren}_\lambda\left(\Gamma, \tau\right)$ where the right-hand side is comprised of renamings $\Gamma \to [x : \tau]$ which are *functions* $\rho : \mathrm{dom}\left([x : \tau]\right) \to \mathrm{dom}\left(\Gamma\right)$ such that $\Gamma\left(\rho\left(x\right)\right) = [x : \tau]\left(x\right) = \tau$. That is, the ρ are functions selecting a variable of type τ in Γ. More concisely, we have an isomorphism $\mathfrak{V}_\tau\left(\Gamma\right) \cong \left\{x \mid (x : \tau) \in \Gamma\right\}$ by which we allow ourselves to consider this a *presheaf of syntax*.

The Yoneda lemma delivers the following insight about a special class of exponential objects in $\widehat{\mathrm{Ren}_\lambda}$. Let $\mathfrak{P} : \widehat{\mathrm{Ren}_\lambda}$. Exponentiation by a representable/variable presheaf, $\mathfrak{V}_\tau = \text{よ}\tau$ gives

$$\begin{aligned}
\mathfrak{P}^{\mathfrak{V}_\Delta}(\Gamma) &= \mathfrak{P}^{\text{ɬ}\tau}(\Gamma) \\
&= \widehat{\mathrm{Ren}_\lambda}\left[\text{ɬ}\Gamma \times \text{ɬ}\tau, \mathfrak{P}\right] && (2) \\
&\cong \widehat{\mathrm{Ren}_\lambda}\left[\text{ɬ}(\Gamma \times \tau), \mathfrak{P}\right] && \text{(since the Yoneda embedding is cartesian closed)} \\
&\cong \mathfrak{P}(\Gamma \times \tau). && \text{(by the Yoneda lemma)}
\end{aligned}$$

Where step (2) follows from the definition of the exponentials in the category of presheaves, c.f. page 46 of [MM92].

The presheaves of typed variables together with the simplified view of variable-exponentiation in $\widehat{\mathrm{Ren}_\lambda}$ will allow us to more easily define algebras for a new algebraic theory $\mathcal{NN}$ of stratified *neutrals and normals*, which allows for a more fine-grained discussion of normal terms.

4.1.2 Stratifying neutral and normal terms

We introduce a new type system over the syntax and type structure of the lambda calculus as defined in Chapter 2. This new type system will distinguish between *neutral terms* and ordinary normal terms. The *neutral terms* are a special class of *normal terms* which are in some sense terms who might otherwise be reducible if not for having a variable as a subterm of an elimination form. Put another way, *neutral terms* can be regarded as those terms which may only be normal because they are waiting on the value of a variable.

Definition 4.1.2 (Neutral and normal judgments)**.**

$$\frac{}{\Gamma \vdash_{\text{ne}} x : \tau}\ (x : \tau) \in \Gamma$$

$$\frac{\Gamma \vdash_{\text{ne}} M : \tau_1 * \tau_2}{\Gamma \vdash_{\text{ne}} \pi_i M : \tau_i} \qquad \frac{\Gamma \vdash_{\text{ne}} t_1 : \tau' \to \tau \qquad \Gamma \vdash_{\text{nf}} t_2 : \tau'}{\Gamma \vdash_{\text{ne}} t_1 t_2 : \tau}$$

$$\Gamma \vdash_{\text{nf}} () : \mathbb{1} \qquad \frac{\Gamma \vdash_{\text{nf}} N_i : \tau_i}{\Gamma \vdash_{\text{nf}} (N_1, N_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma, x : \tau \vdash_{\text{nf}} b : \tau'}{\Gamma \vdash_{\text{nf}} \lambda x : \tau.\, b : \tau \to \tau'}$$

$$\frac{\Gamma \vdash_{\text{ne}} t : \theta}{\Gamma \vdash_{\text{nf}} t : \theta}\ (\theta \in T \text{ A BASE TYPE})$$

Definition 4.1.3 (Algebraic theory of neutrals and normals)**.** The judgments above suggest the definition of a theory with a richer sort structure than that of the theory $\mathcal{L}_{\lambda\alpha\beta\eta}$ defined back in Chapter 2. In particular, the new sort structure will feature both a neutral sort and a normal sort for each type τ, allowing us to define algebras that distinguish between these.

We define $\mathcal{NN}$ as the algebraic theory corresponding (c.f. Theorem 1.2.9) to the category with the following objects and generating morphisms:

1. The objects are the collection generated by taking (syntactic) products and exponentials over the collection $\Sigma = \left\{\mathcal{M}_\tau \mid \tau \in \widetilde{T}\right\} \cup \left\{\mathcal{N}_\tau \mid \tau \in \widetilde{T}\right\} \cup \left\{\mathcal{V}_\tau \mid \tau \in \widetilde{T}\right\}$

2. Generating morphisms, for each $\tau, \tau' \in \widetilde{T}$ and each base type $\theta \in T$:

$$\begin{aligned}
\mathrm{var}_\tau &: \mathcal{V}_\tau \to \mathcal{M}_\tau \\
\mathrm{fst}_\tau^{\tau'} &: \mathcal{M}_{\tau\times\tau'} \to \mathcal{M}_\tau \\
\mathrm{snd}_\tau^{\tau'} &: \mathcal{M}_{\tau'\times\tau} \to \mathcal{M}_\tau \\
\mathrm{app}_\tau^{\tau'} &: \mathcal{M}_{\tau'\to\tau} \times \mathcal{N}_{\tau'} \to \mathcal{M}_\tau \\
\mathrm{incl}_\theta &: M_\theta \to N_\theta \\
\mathrm{unit} &: \mathbb{1} \to \mathcal{N}_{\mathbb{1}} \\
\mathrm{pair}_\tau^{\tau'} &: \mathcal{N}_\tau \times \mathcal{N}_{\tau'} \to \mathcal{N}_{\tau\times\tau'} \\
\mathrm{abs}_{\tau\to\tau'} &: {\mathcal{N}_{\tau'}}^{\mathcal{V}_\tau} \to \mathcal{N}_{\tau\to\tau'}.
\end{aligned}$$

3. The laws are the usual $\alpha-$, $\beta-$, and $\eta-$rules.

Note that the domain of the unit operation symbol is the nullary product of sorts.

Remark 4.1.4. Any $\widehat{\mathrm{Ren}_\lambda}-$ lambda algebra over the family $\{\mathfrak{X}_\tau\}_{\tau\in\widetilde{T}}$ gives rise to an algebra for the algebraic theory $\mathcal{NN}$ over the family $\{\mathfrak{A}_\tau\}_{\tau\in\widetilde{T}}$ by setting

$$\begin{aligned}
\mathfrak{A}_{\mathcal{M}_\tau} &= \mathfrak{X}_\tau \\
\mathfrak{A}_{\mathcal{N}_\tau} &= \mathfrak{X}_\tau \\
\mathfrak{A}_{\mathcal{V}_\tau} &= \mathfrak{V}_\tau
\end{aligned}$$

and setting the operation symbols as in the lambda algebra.

Remark 4.1.5 (A lambda algebra of open substitutions)**.** By Theorem 1.5.11, the syntactic category Cn_λ of the $\alpha\beta\eta$-lambda calculus has a lambda algebra, and an induced interpretation of terms $[\![-]\!]$.

The morphisms

$$\begin{aligned}
\pi_1 &: [\![\tau]\!] \times [\![\tau']\!] \to [\![\tau]\!] \\
\pi_2 &: [\![\tau]\!] \times [\![\tau']\!] \to [\![\tau']\!] \\
\epsilon &: [\![\tau']\!]^{[\![\tau]\!]} \times [\![\tau]\!] \to [\![\tau']\!]
\end{aligned}$$

in the syntactic category can be lifted by $\mathfrak{Tm}$ to $\widehat{\mathrm{Ren}_\lambda}$ to provide the operations for a lambda algebra over the family $\left\{\mathfrak{Tm}\,(\tau)\right\}_{\tau\in\widetilde{T}}$

The operations are given as follows:

$$\begin{gathered}
\mathfrak{V}_\tau \xrightarrow{[\![-]\!]} \mathfrak{Tm}\,(\tau) \\
\mathbb{1} \xrightarrow{i_{\mathbb{1}}} \mathfrak{Tm}\,(\mathbb{1}) \\
\mathfrak{Tm}\left(\tau \times \tau'\right) \xrightarrow{\mathfrak{Tm}(\pi_1)} \mathfrak{Tm}\,(\tau) \\
\mathfrak{Tm}\left(\tau \times \tau'\right) \xrightarrow{\mathfrak{Tm}(\pi_2)} \mathfrak{Tm}\left(\tau'\right) \\
\mathfrak{Tm}\,(\tau) \times \mathfrak{Tm}\left(\tau'\right) \xrightarrow{i_\pi} \mathfrak{Tm}\left(\tau \times \tau'\right) \\
\mathfrak{Tm}\left(\tau'^{\tau}\right) \times \mathfrak{Tm}\,(\tau) \xrightarrow{i_\pi} \mathfrak{Tm}\left(\left(\tau'\right)^{\tau} \times \tau\right) \xrightarrow{\mathfrak{Tm}(\epsilon)} \mathfrak{Tm}\left(\tau'\right) \\
\mathfrak{Tm}\left(\tau'\right)^{\mathfrak{V}_\tau} \xrightarrow{\cong} \mathfrak{Tm}\left(\left(\tau'\right)^{\tau}\right)
\end{gathered}$$

where $i_{\mathbb{1}}$ and i_π are isomorphisms induced by $\mathfrak{Tm}$'s status as a Cartesian closed functor. We write $\mathfrak{Tm}$ for this algebra.

Remark 4.1.6. The lambda algebra just defined induces, by Remark 4.1.4, an $\mathcal{NN}$-algebra over $\mathfrak{Tm}$ with the assignment of sorts

$$\begin{aligned}
\mathcal{V}_\tau &\mapsto \mathfrak{V}_\tau \\
\mathcal{M}_\tau &\mapsto \mathfrak{Tm}\,(\tau) \\
\mathcal{N}_\tau &\mapsto \mathfrak{Tm}\,(\tau)
\end{aligned}$$

and letting the operations be exactly those of the lambda algebra: since the $\mathcal{M}_\tau = \mathcal{N}_\tau = \mathfrak{Tm}\,(\tau)$, the signatures of the required operations are exactly the same. We write $(\mathfrak{Tm}, \mathfrak{Tm})$ for this induced algebra.

4.1.3 Presheaves of syntax over the category of renamings

Definition 4.1.7 (A presheaf of open syntactic terms)**.** For each type $\tau \in \widetilde{T}$ and context $\Gamma \in \mathrm{Ren}_\lambda$, define

$$\mathfrak{L}_\tau\,(\Gamma) = \left\{t \mid \Gamma \vdash t : \tau\right\}.$$

Together with the *renaming action*, defined for each $\rho : \Gamma' \to \Gamma$ as

$$\begin{aligned}
\rho^* : \left\{t \mid \Gamma \vdash t : \tau\right\} &\to \left\{t \mid \Gamma' \vdash t : \tau\right\} \\
t &\mapsto \rho^* t
\end{aligned}$$

where $\rho^* t$ is the result of ρ (regarded as a substitution) acting on t by the action of the clone model (c.f. Theorem 1.2.7), the assignment above in fact defines a presheaf

$$\mathfrak{L}_\tau : \widehat{\mathrm{Ren}_\lambda}$$

where functorialty of the renaming action is inherited from that of the clone model as in Theorem 1.2.7.

Remark 4.1.8 (A lambda algebra on the presheaves of open syntax)**.** The presheaves of open syntax $\mathfrak{L}_\tau$ form the objects of an algebra for the theory $\mathcal{L}_{\lambda\alpha\beta\eta}$ defined in Definition 2.3.3.

The operations are given by the usual typing rules for the simply typed lambda calculus, as in Definition 2.1.3. We write $\mathfrak{L}$ for this algebra.

Remark 4.1.9 (A stratified-lambda algebra on the presheaves of open syntax)**.** By Remark 4.1.4, the lambda algebra of open syntax from Definition 4.1.8 induces an algebra of the theory $\mathcal{NN}$.

We can define presheaves of neutral and normal terms similarly, with the same presheaf action by renaming as before.

Definition 4.1.10 (A presheaf of neutral terms)**.** For each $\tau \in \widetilde{T}$, the assignment

$$\mathfrak{Ne}_\tau\left(\Gamma\right) = \left\{t \mid \Gamma \vdash_{\mathrm{ne}} t : \tau\right\}$$

for each context $\Gamma : \mathrm{Ren}_\lambda$ defines a presheaf

$$\mathfrak{Ne}_\tau : \widehat{\mathrm{Ren}_\lambda}$$

under the renaming action.

Definition 4.1.11 (A presheaf of normal terms)**.** For each $\tau \in \widetilde{T}$, the assignment

$$\mathfrak{Nf}_\tau = \left\{t \mid \Gamma \vdash_{\mathrm{nf}} t : \tau\right\}_{\Gamma \in \mathrm{Ren}_\lambda}$$

for each context $\Gamma : \mathrm{Ren}_\lambda$ defines a presheaf

$$\mathfrak{Nf}_\tau : \widehat{\mathrm{Ren}_\lambda}$$

under the renaming action.

After teasing out the latent binding structure enjoyed by the category of presheaves, we will find in Definition 4.1.13 that the presheaves of neutrals and normals give rise to a $\mathcal{NN}$-algebra over $\widehat{\mathrm{Ren}_\lambda}$.

Remark 4.1.12 (Why the syntactic category just won't do)**.** As mentioned earlier, a major motivation for defining the category of renamings in the first place (to which we went to considerable trouble!) is that the syntactic category is an unsuitable base category over which to define presheaves like the ones above. Let's go back to basics and recall that a presheaf is not just a fancy family of sets, but crucially comes with a contravariant action taking a morphism in the base category to a function of *sets of the family* going in the opposite direction. If we take the example of the presheaf of opens of type τ, $\mathfrak{L}_\tau$, there are no apparent problems. We could define the family $\left\{\mathfrak{L}_\tau\left(\Delta\right)\right\}_{\Delta \in \mathrm{Cn}_\lambda}$ essentially as before, and use the following presheaf action:

$$\begin{gathered}\mathrm{Cn}_\lambda\left(\Delta, \Gamma\right) \to \mathfrak{Set}\left(\mathfrak{L}_\tau\left(\Gamma\right), \mathfrak{L}_\tau\left(\Delta\right)\right) \\ \delta \mapsto \left(t \mapsto \delta^* t\right)\end{gathered}$$

which acts by actually performing the substitution on a term in the set. This action's utility breaks down however, when considering the presheaves $\mathfrak{Nf}_\tau$. Supposing we were able to define $\mathfrak{Nf}_\tau$ instead over the classifying category, with the same family as before but with the natural substitution action shown above we would have for each substitution $\delta : \Delta \to \Gamma$ a function of sets $\mathfrak{Nf}_\tau(\Gamma) \to \mathfrak{Nf}_\tau(\Delta)$. For a variable $(x : \tau) \in \Gamma$, then, the substitution $\left[(\lambda x : \mathbb{1}.\, x)\,\mathrm{unit}/y\right]$ is taken to a function of sets $\mathfrak{Nf}_\tau(\Gamma, y : \mathbb{1}) \to \mathfrak{Nf}_\tau(\Gamma)$ performing that substitution on terms. The term $(\lambda x.\, x)\,\mathrm{unit}$ is evidently not a normal form (since a β-reduction step applies) but x is a normal form in $\mathfrak{Nf}_\tau(\Gamma, y : \mathbb{1})$ so that this action forces $(\lambda x : \mathbb{1}.\, x)\,\mathrm{unit} \in \mathfrak{Nf}_\tau(\Gamma)$. The essence of the problem is that normal forms are not closed under substitutions, so the natural substitution action is ruled out. Whether there is a different suitable action by the classifying category on this family is a good question to ponder.

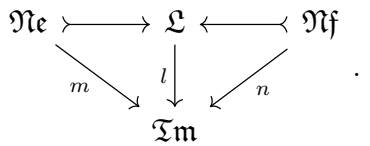
4.1.4 A stratified lambda-algebra of neutrals and normals in Renhat

Definition 4.1.13. The presheaves of neutrals and normals defined above give rise to an algebra of the theory $\mathcal{NN}$ in $\widehat{\mathrm{Ren}_\lambda}$. The sorts $\mathcal{N}_\tau$ (resp. $\mathcal{M}_\tau$) are taken to be the presheaves $\mathfrak{Nf}_\tau$ (resp. $\mathfrak{Ne}_\tau$) and the sorts $\mathcal{V}_\tau$ are taken to be the representable/variable presheaves $\mathfrak{V}_\tau$ defined in Definition 4.1.1.

The operations correspond to the typing rules of Definition 4.1.2.

We write $(\mathfrak{Ne}, \mathfrak{Nf})$ for this algebra.

Remark 4.1.14. It turns out that the algebra $(\mathfrak{Ne}, \mathfrak{Nf})$ is initial in the category $\mathrm{Mod}_{\widehat{\mathrm{Ren}_\lambda}}(\mathcal{NN})$ [Fio02]. With this fact, the $\mathcal{NN}$-algebra $(\mathfrak{Tm}, \mathfrak{Tm})$ over the family $\left\{\left(\mathfrak{Tm}(\tau), \mathfrak{Tm}(\tau)\right)\right\}_\tau$ together with the just presented $\mathcal{NN}$-algebra over the family $\left\{(\mathfrak{Ne}_\tau, \mathfrak{Nf}_\tau)\right\}_{\tau \in \widetilde{T}}$ and the $\mathcal{NN}$-algebra over the family $\{\mathfrak{L}_\tau\}_{\tau \in \widetilde{T}}$ induce, for each $\tau \in \widetilde{T}$, $\mathcal{NN}$-homomorphisms $l : \mathfrak{L} \to \mathfrak{Tm}$ and $(m, n) : (\mathfrak{Ne}, \mathfrak{Nf}) \to (\mathfrak{Tm}, \mathfrak{Tm})$ such that the following diagram commutes:



In explicit terms, the map l_τ for each $\tau \in \widetilde{T}$ is the usual semantic interpretation of terms in the syntactic category:

$$\begin{aligned} l_\tau(\Gamma) : \mathfrak{L}_\tau(\Gamma) &\to \mathfrak{Tm}(\tau)(\Gamma) \\ t &\mapsto [\![\Gamma \vdash t : \tau]\!]. \end{aligned}$$

The maps m_τ, n_τ are the usual semantic interpretation precomposed with the respective inclusions into open terms.

4.2 Desiderata for a normalization function

For all this talk of normals forms and normalization, we haven't said much about what a normalization function must be. Of course, it should take a lambda term to a normal form, but the desiderata are far more than that. The following are some of the important properties a normalization function $\mathrm{nf}_\tau^\Gamma : \mathfrak{L}_\tau(\Gamma) \to \mathfrak{Nf}_\tau(\Gamma)$ should satisfy:

- Semantics preservation: For all terms $t \in \mathfrak{L}_\tau(\Gamma)$,

$$\mathrm{nf}_\tau^\Gamma(t) \equiv_{\alpha\beta\eta} t.$$

- Equation preservation: For all terms $t, t' \in \mathfrak{L}_\tau(\Gamma)$,

$$t \equiv_{\beta\eta} t' \Rightarrow \mathrm{nf}_\tau^\Gamma(t) \equiv_{\alpha\beta\eta} \left(t'\right).$$

- Equation reflection: For all terms $t, t' \in \mathfrak{L}_\tau(\Gamma)$,

$$\mathrm{nf}_\tau^\Gamma(t) = \mathrm{nf}_\tau^\Gamma\left(t'\right) \Rightarrow t \equiv_{\alpha\beta\eta} t'$$

These properties together with the signature of the function above encode most of what one might reasonably expect of a normalization function, and indeed these are the ones we shall prove about the normalization we plan to construct. The strength of the approach we have built towards so far is that the proofs of the above properties will essentially fall out of having constructed the normalization function as a composite (with the right domain and codomain) of some glued maps. It should be said that the above are not a complete enumeration of the properties one might request of a normalization function. Some theoreticians might demands that normal forms are *fixed* by the normalization function; that is, $\mathrm{nf}_\tau^\Gamma(N) = N$ for any normal form N. We do not show this; for that, refer the reader to Fiore [Fio02].

Now that we understand what a normalization function must be, we can set about constructing it. As hinted at above, our course will be plotted like this:

First, we will elaborate the cartesian closed structure of the gluing category so that we can extend its lambda algebras along that structure to interpretations of lambda terms. Later, we will define an $\mathcal{NN}$-algebra of *glued* neutrals and normals whose neutral objects μ_θ will serve as the interpretation of a base type θ. Finally, we will define the normalization function as a composite of the interpretation induced by this assignment and (more or less) some of the glued operations for the $\mathcal{NN}$-algebra mentioned above. It will turn out that the characterizing property of such glued morphisms says exactly that the semantics is preserved by these morphisms, from which the first property is almost trivial. The second property is even more immediate, as we will find. [TODO: do another pass of this intro]

Definition 4.2.1 (Forgetful projections)**.** The assignments

$$\begin{aligned} \mathrm{Gl}_\lambda &\to \mathrm{Cn}_\lambda \\ (R, q, \Delta) &\mapsto \Delta \end{aligned}$$

$$\begin{aligned} \mathrm{Gl}_\lambda \left((R, q, \Delta), (R', q', \Delta') \right) &\to \mathrm{Cn}_\lambda (\Delta, \Delta') \\ (d, \delta) &\mapsto \delta \end{aligned}$$

form a functor $\pi_{\mathrm{sem}} : \mathrm{Gl}_\lambda \to \mathrm{Cn}_\lambda$ which *forgets the syntax* leaving only the semantic components of a glued object or glued morphism.

We similarly get a functor $\pi_{\mathrm{syn}} : \mathrm{Gl}_\lambda \to \widehat{\mathrm{Ren}_\lambda}$ which strips away the semantics leaving only the syntax defined by

$$\begin{aligned} (R, q, \Delta) &\mapsto R \\ (d, \delta) &\mapsto d. \end{aligned}$$

Remark 4.2.2. The forgetful projection π_{sem} is cartesian closed, in the sense of Remark 3.4.1, as the reader may verify.

4.3 Cartesian closed structure for the gluing category

In order to give an interpretation of terms in the gluing category, we must give an explicit account of its cartesian closed structure. We will say what its products are, but not prove the universal property: we send the reader to [SS18] for that part. We will however give a sketch of the universal property for the exponentials, which is not shown in either of the sources we consulted on connections between gluing and normalization [SS18] [Fio02]. The result itself is well-known[1], but the sketch has not been fully verified and should be taken with a grain of salt.

Theorem 4.3.1 (Products in the gluing category)**.** The terminal object (nullary product) is $\left(\mathbb{1} : \widehat{\mathrm{Ren}_\lambda}, t : \mathbb{1} \to \mathfrak{Tm}(\mathbb{1}), \mathbb{1} : \mathrm{Cn}_\lambda \right)$ where t is the unique map $\mathbb{1} \xrightarrow[\cong]{} \mathfrak{Tm}(\mathbb{1})$ guaranteed by cartesian closure of $\mathfrak{Tm}$. The binary product $(P, p, \Delta) \times (Q, q, \Gamma)$ of (P, p, Δ) and (Q, q, Γ) is $(P \times Q, r, \Delta \times \Gamma)$ where r is the composite

$$P \times Q \xrightarrow{p \times q} \mathfrak{Tm}(\Delta) \times \mathfrak{Tm}(\Gamma) \xrightarrow[\cong]{i_\pi} \mathfrak{Tm}(\Delta \times \Gamma).$$

Theorem 4.3.2 (Exponentials in the gluing category)**.** For objects (R_1, q_1, Δ_1) and (R_2, q_2, Δ_2) in Gl_λ, the exponential $(R_2, q_2, \Delta_2)^{(R_1, q_1, \Delta_1)}$ is $\left(R, q, {\Delta_2}^{\Delta_1} \right)$ where R and

[1] See e.g. Johnstone's Elephant [Joh02] where it is posed as an exercise.

q are defined by the *pullback*

$$\begin{array}{ccc} R & \xrightarrow{\quad r \quad} & {R_2}^{R_1} \\ {\scriptstyle q}\downarrow & & \downarrow{\scriptstyle q_{2*}} \\ \mathfrak{Tm}\left(\Delta_2^{\Delta_1}\right) & \xrightarrow[q_1{}^* \circ \tilde{p}]{} & \mathfrak{Tm}\left(\Delta_2\right)^{R_1} \end{array}$$

where p is the composite

$$\mathfrak{Tm}\left(\epsilon\right) \circ \left(\mathfrak{Tm}\left(\Delta_2^{\Delta_1}\right) \times \mathfrak{Tm}\left(\Delta_1\right) \overset{\cong}{\to} \mathfrak{Tm}\left(\Delta_2^{\Delta_1} \times \Delta_1\right)\right).$$

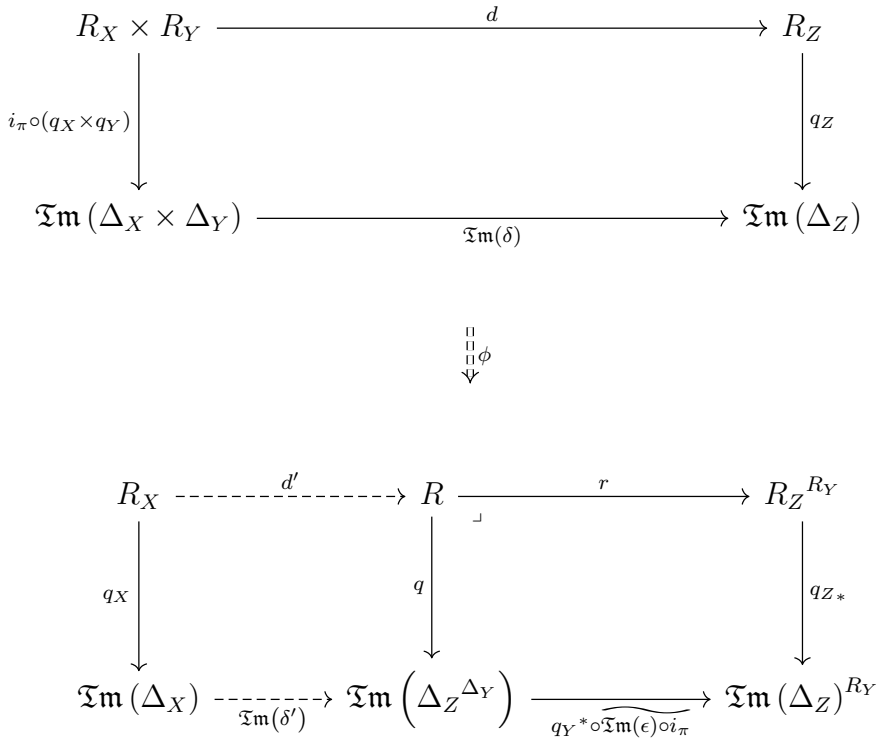
Sketch. The universal property for exponentials is encoded by the product-hom adjunction so that it suffices, by the definition of products in the gluing category, to show an isomorphism

$$\phi : P \cong E : \psi$$

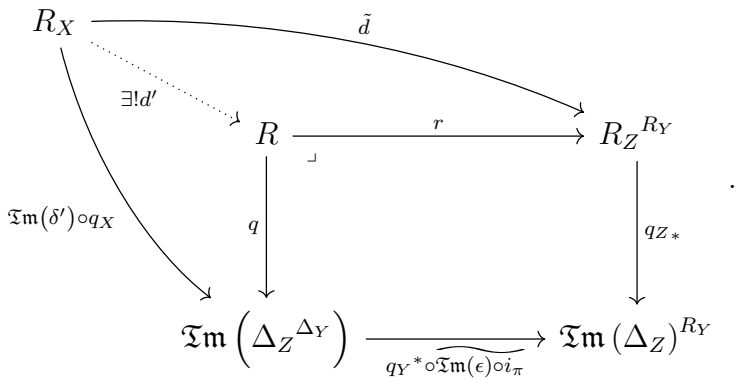
where

$$\begin{aligned} &\mathfrak{Tm}\left(\Delta_2^{\Delta_1}\right) \times \mathfrak{Tm}\left(\Delta_1\right) \xrightarrow{i_\pi} \mathfrak{Tm}\left(\Delta_2^{\Delta_1} \times \Delta_1\right) \\ &P = \mathrm{Gl}_\lambda\left(\left(R_X \times R_Y, i_\pi \circ \left(q_X \times q_Y\right), \Delta_X \times \Delta_Y\right), \left(R_Z, q_Z, \Delta_Z\right)\right) \\ &E = \mathrm{Gl}_\lambda\left(\left(R_X, q_X, \Delta_X\right), \left(R_Z, q_Z, \Delta_Z\right)^{\left(R_Y, q_Y, \Delta_Y\right)}\right). \end{aligned}$$

To come up with this isomorphism, it helps to consider what the morphisms in P and E are. Looking back to the definition of products and our proposed definition for the exponentials reveals that a morphism $(d, \delta) \in P$ must fit into the upper half of the diagram below, while any morphism $(d', \delta') \in E$ must fit into the left side of the lower half of the diagram. As such, the components of the map ϕ evaluated at some $(d, \delta) \in P$ must be the dotted arrows in the lower half of the diagram below:



Observing the arrows in sight, a plausible choice for δ' is $\tilde{\delta}$. The choice for d' is a more tricky question. We know that d' should be an arrow into the pullback R. Just about the only thing we know about R is that it fits into the pullback diagram above. Our relatively narrow knowledge of R turns out to be exactly what clears the air, making our choice of d' automatic completely: d' must be, can only be, one of the mediators required by the universal property of the pullback, namely the dotted arrow in the diagram below. To get this dotted arrow, we need to verify that the outer square of the following diagram commutes:



We need to show that

$$q_Y^* \circ \widetilde{\mathfrak{Tm}(\epsilon) \circ i_\pi} \circ \mathfrak{Tm}\left(\tilde{\delta}\right) \circ q_X = q_{Z*} \circ \tilde{d},$$

which should strike the reader as suspiciously similar to the characterizing property of the glued morphism (d, δ), namely that

$$\mathfrak{Tm}(\delta) \circ i_\pi \circ (q_X \times q_Y) = q_Z \circ d.$$

In what follows, we call this equation the gluing equation.

To untangle this mess, we pass to the lambda notation for the transpose as in Chapter 2. We can express the left-hand side of the desired equality in this notation (dropping the explicit type annotations and adding explicit variables for both abstracted variables and ordinary free variables representing morphism inputs[2]) as

$$\begin{aligned}
& q_Y^* \circ \left(\lambda y.\, \mathfrak{Tm}(\epsilon)\left(i_\pi(f, y)\right)\right) \circ \mathfrak{Tm}\left(\tilde{\delta}\right)(x) \circ q_X \\
&= q_Y^* \circ \left(\lambda y.\, \mathfrak{Tm}(\epsilon)\left(i_\pi\left(\mathfrak{Tm}\left(\delta(x)\right), y\right)\right)\right) \circ q_X \\
&= q_Y^* \circ \left(\lambda y.\, \mathfrak{Tm}(\epsilon)\left(i_\pi\left(\mathfrak{Tm}\left(\tilde{\delta}(x)\right), y\right)\right)\right) \circ q_X \\
&= q_Y^* \circ \left(\lambda y.\, \mathfrak{Tm}(\delta)\left(i_\pi(x, y)\right)\right) \circ q_X \\
&= \lambda y.\, \mathfrak{Tm}(\delta)\left(i_\pi\left(q_X(x), q_Y(y)\right)\right)
\end{aligned}$$

where the second and third steps respectively follow from associativity of composition and the lifting along $\mathfrak{Tm}$ of the universal property of transposes in Cn_λ. We are also using the fact that the universal property of exponentials forces i_π to be the unique isomorphism which acts as a syntactic pair constructor, i.e., it takes genuine pairs to syntactic pairs. What we are left with is exactly the transpose of the left-hand side of the gluing equation:

$$\begin{aligned}
\overbrace{\mathfrak{Tm}(\delta) \circ i_\pi \circ (q_X \times q_Y)} &= \lambda y.\, \mathfrak{Tm}(\delta)\left(i_\pi\left((q_X \times q_Y)(x, y)\right)\right) \\
&= \lambda y.\, \mathfrak{Tm}(\delta)\left(i_\pi\left(q_X(x), q_Y(y)\right)\right).
\end{aligned}$$

Moreover, the right-hand side of the desired equality is evidently the transpose of the right-hand side of the gluing equation. Because taking transposes is well-defined, the preceding two facts gives us the desired equality.

We can now summarize the rightward map for the isomorphism:

$$\begin{aligned}
&\phi_1(d) = d' && \text{(the unique mediating arrow from the discussion above)} \\
&\phi_2(\delta) = \tilde{\delta}.
\end{aligned}$$

The leftward side of the isomorphism, ψ, is substantially easier, because we don't need to interact much with the pullback. We define

[2]For the sake of clarity, we will not do this unless necessary. Our use of free variables to represent morphism inputs is meant to mirror how we handled the inputs to operation symbols.

$$\psi_1\left(d'\right) = \epsilon \circ \left(\left(d' \circ r\right) \times \mathrm{id}\right)$$
$$\psi_2\left(\delta'\right) = \epsilon \circ \left(\delta' \times \mathrm{id}\right),$$

where the former epsilon is the evaluation map in $\widehat{\mathrm{Ren}_\lambda}$ and the latter epsilon is the evaluation map in Cn_λ.

With our maps in hand, we can verify that they form a bijection. Because we have defined d' by the universal property of the pullback, we have that $d' \circ r = \tilde{d}$ whence $\psi_1\left(\phi_1\left(d\right)\right) = \epsilon \circ \left(\tilde{d} \times \mathrm{id}\right)$ so that $\psi_1\left(\psi_1\left(d\right)\right) = d$ by the universal property of the transpose in $\widehat{\mathrm{Ren}_\lambda}$. The second required equality also follows by the universal property of the transpose, but this time using the universal property of the transpose *in* Cn_λ. We have $\psi_2\left(\phi_2\left(\delta\right)\right) = \epsilon \circ \left(\tilde{\delta} \times \mathrm{id}\right)$ so that $\psi_2\left(\phi_1\left(\delta\right)\right) = \delta$ as required.

With that, we have sketched the bijection. The full details, along with the naturality condition are left to the reader. ∥

Corollary 4.3.3. The gluing category Gl_λ is cartesian closed.

4.4 Gluing syntax to semantics

We will define some glued objects which, in some sense, glue the presheaves of syntax we have defined in $\widehat{\mathrm{Ren}_\lambda}$ together with their semantics in the classifying category. These glued objects will ultimately assemble into the objects for an algebra of the theory NN.

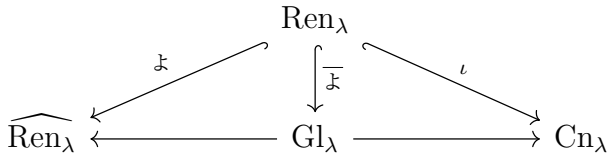
Definition 4.4.1. The relative hom functor induces the embedding

$$\overline{\text{ﻉ}} : \mathrm{Ren}_\lambda \hookrightarrow \mathrm{Gl}_\lambda$$
$$\Gamma \mapsto \left(\text{ﻉ}\left(\Gamma\right), \text{ﻉ}\left(\Gamma\right) \xrightarrow{\iota_\Gamma} \mathfrak{Tm}\left(\Gamma\right), \Gamma\right)$$

where

$$\left(\underline{\iota}_\Gamma\right)_\Delta : \mathrm{Ren}_\lambda\left(\Delta, \Gamma\right) \to \mathrm{Cn}_\lambda\left(\Delta, \Gamma\right)$$
$$\rho \mapsto \iota\left(\rho\right).$$

This fits into the following diagram



$$R \xleftarrow{\pi_{\mathrm{syn}}} (R, q, \Delta) \xrightarrow{\pi_{\mathrm{sem}}} \Delta$$

and satisfies the following form of the Yoneda lemma:

$$(d, \delta) \longmapsto d\,(\mathrm{id})$$

$$\mathrm{Gl}_\lambda\left(\overline{\text{よ}}\,(-), (R, q, \Delta)\right) \xrightarrow{\cong} R\,(-)\ .$$

$$\pi_{\mathrm{sem}} \searrow \qquad \swarrow q$$

$$\mathrm{Cn}_\lambda\left(\iota\,(-), \Delta\right) = \mathfrak{Tm}\,(\Delta)$$

This embedding allows us to define a typed-indexed family of glued objects which glue the syntax, $x, y, z, \ldots$, etc., of variables to the their semantics as substitutions.

Definition 4.4.2 (Gluing syntax and semantics of variables)**.** We define the glued object

$$\nu_\tau = \overline{\text{よ}}\,(\tau) = \left(\mathfrak{V}_\tau, \mathfrak{V}_\tau \to \mathfrak{Tm}\,(\tau), \tau\right)$$

where the components of the interpretation are the usual interpretation of terms in the classifying category.

We do the same for the syntactic presheaves of neutrals and normals we defined earlier. In each case, the components of the quotient map are the interpretation of terms in the classifying category.

Definition 4.4.3 (Gluing syntax and semantics of neutrals)**.** We define

$$\mu_\tau = \left(\mathfrak{Ne}_\tau, \mathfrak{Ne}_\tau \xrightarrow{m_\tau} \mathfrak{Tm}\,(\tau), \tau\right).$$

Definition 4.4.4 (Gluing syntax and semantics of normals)**.** We define

$$\eta_\tau = \left(\mathfrak{Nf}_\tau, \mathfrak{Nf}_\tau \xrightarrow{n_\tau} \mathfrak{Tm}\,(\tau), \tau\right).$$

With these glued objects, we can define an $\mathcal{NN}$-algebra over these families of glued variables, glued neutrals, and glued normals. In turn Definition 1.5.10 will allow us to leverage this to interpret substitutions as gluing category morphisms between these glued objects. The algebra we define will be constructed by gluing

together the syntactic $\mathcal{NN}$-algebra in $\widehat{\mathrm{Ren}_\lambda}$ together with the semantic $\mathcal{NN}$-algebra[3] in the classifying category.

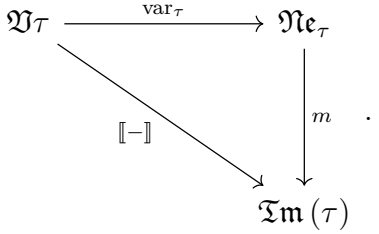
Definition 4.4.5 (An algebra of stratified neutrals and normals in the gluing category)**.** The family $\left\{(\mu_\tau, \eta_\tau)\right\}_{\tau \in \widetilde{T}}$ define the objects of an $\mathcal{NN}$-algebra with the operations given as follows:

- For each $\tau \in \widetilde{T}$, the pair of maps

  $$\left(\mathrm{var}_\tau : \mathfrak{V}_\tau \to \mathfrak{Ne}_\tau, \mathrm{id}_{[\![\tau]\!]}\right)$$

  is a map $\nu_\tau \to \mu_\tau$ in Gl_λ.

  That this pair forms a glued morphism follows from the fact that $(m, n) : (\mathfrak{Ne}, \mathfrak{Nf}) \to (\mathfrak{Tm}, \mathfrak{Tm})$ is a NN-homomorphism. In particular, we have that the following diagram commutes:

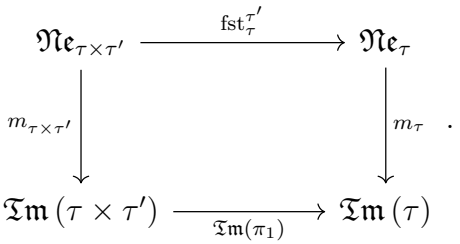
  Inserting the lifted identity map at the bottom gives precisely the required square.

- For each $\tau, \tau' \in \widetilde{T}$, the pair of maps

  $$\left(\mathrm{fst}_\tau^{\tau'} : \mathfrak{Ne}_{\tau \times \tau'} \to \mathfrak{Ne}_\tau, \pi_1 : [\![\tau]\!] \times [\![\tau']\!] \to [\![\tau]\!]\right)$$

  is a map $\mu_{\tau \times \tau'} \to \mu_\tau$ in Gl_λ.

  That this pair forms a glued morphisms again follows from the fact that $(m, n) : (\mathfrak{Ne}, \mathfrak{Nf}) \to (\mathfrak{Tm}, \mathfrak{Tm})$ is a NN-homomorphism. In particular, we have that the following diagram commutes:

  Which is exactly the required square. In fact, each of the remaining cases work out in this unsurprising way, so we will stop mentioning it. It is worth noting

[3]The algebra referred to here is the one induced by upgrading the usual lambda algebra in the classifying category according to Remark 4.1.4

that one reason these squares line up so cleanly is that we defined the lambda algebra on $\mathfrak{Tm}$ by lifting the relevant operations from Cn_λ into $\widehat{\mathrm{Ren}_\lambda}$ along $\mathfrak{Tm}$.

- For each $\tau, \tau' \in \widetilde{T}$, the pair of maps
$$\left(\mathrm{snd}_\tau^{\tau'} : \mathfrak{Ne}_{\tau' \times \tau} \to \mathfrak{Ne}_\tau, \pi_2 : [\![\tau']\!] \times [\![\tau]\!] \to [\![\tau]\!]\right)$$
is a map $\mu_{\tau' \times \tau} \to \mu_\tau$ in Gl_λ.
- For each $\tau, \tau' \in \widetilde{T}$, the pair of maps
$$\left(\mathrm{app}_\tau^{\tau'} : \mathfrak{Ne}_{\tau' \to \tau} \times \mathfrak{Nf}_{\tau'} \to \mathfrak{Ne}_\tau, \epsilon : \left([\![\tau']\!] \to [\![\tau]\!]\right) \times [\![\tau']\!] \to [\![\tau]\!]\right)$$
is a map $\mu_{\tau' \to \tau} \times \eta_{\tau'} \to \mu_\tau$ in Gl_λ.
- For each base type $\theta \in T$, the pair of isomorphisms
$$\left(\mathfrak{Ne}_\theta \cong \mathfrak{Nf}_\theta, \mathrm{id}_{[\![\theta]\!]}\right)$$
is an isomorphism $\mu_\theta \cong \eta_\theta$ in Gl_λ.
- The pair of isomorphisms
$$\left(\mathbb{1} \cong \mathfrak{Nf}_{\mathbb{1}}, \mathrm{id}_{\mathbb{1}}\right)$$
is an isomorphism $\mathbb{1} \cong \eta_{\mathbb{1}}$ in Gl_λ.
- For $\tau, \tau' \in \widetilde{T}$, the pair of isomorphisms
$$\left(\mathrm{pair}_{\tau \times \tau'} : \mathfrak{Nf}_\tau \times \mathfrak{Nf}_{\tau'} \xrightarrow{\cong} \mathfrak{Nf}_{\tau \times \tau'}, \mathrm{id}_{[\![\tau]\!] \times [\![\tau']\!]}\right)$$
is an isomorphism $\eta_\tau \times \eta_{\tau'} \xrightarrow{\cong} \eta_{\tau \times \tau'}$ in Gl_λ.
- For $\tau, \tau' \in \widetilde{T}$, the pair of isomorphisms
$$\left(\mathrm{abs}_{\tau \to \tau'} : \mathfrak{Nf}_{\tau'}^{\mathfrak{V}_\tau} \xrightarrow{\cong} \mathfrak{Nf}_{\tau \to \tau'}, \mathrm{id}_{[\![\tau']\!]^{[\![\tau]\!]}}\right)$$
is an isomorphism $\eta_{\tau'}^{\nu_\tau} \xrightarrow{\cong} \eta_{\tau \to \tau'}$ in Gl_λ.

Note that the glued operations are given by pairs of syntactic operations and *semantic operations* for elimination forms, and pairs of syntactic operations and the *identity* for introduction forms.

Finally we can give an interpretation of base types in the gluing category in terms of the glued neutrals by the assignment

$$\begin{aligned} T &\xrightarrow{\bar{s}} \mathrm{Gl}_\lambda \\ \theta &\mapsto \mu_\theta. \end{aligned}$$

Together with a suitable interpretation of base constructors and eliminators (which are taken to be a parameter of this development) as glued morphisms, this interpretation defines a lambda algebra of glued neutrals. The universal property of the classifying category and its lambda algebra in turn induces a unique functor

$$[\![-]\!]_{\mathrm{Gl}_\lambda} : \mathrm{Cn}_\lambda \to \mathrm{Gl}_\lambda$$

as in Theorem 1.5.11. We write $[\![-]\!]_{\mathrm{Gl}_\lambda} : \mathrm{Cn}_\lambda \to \mathrm{Gl}_\lambda$ for both this functor and the interpretation of terms acquired by precomposing the functor with the usual interpretation of terms in the classifying category. It is crucial to understand that though we have $[\![\theta]\!]_{\mathrm{Gl}_\lambda} = \mu_\theta$ for base types θ, this equality does *not* hold at higher types $\tau \in \widetilde{T}$. In the case of products, the interpreted type $[\![\theta \times \theta']\!]_{\mathrm{Gl}_\lambda}$ is the actual product $\mu_\theta \times \mu_{\theta'}$ and not $\mu_{\theta \times \theta'}$. In what follows, we still allow ourselves to write $[\![-]\!]$ for the usual interpretation of terms in the classifying category, distinguishing the new interpretation only by a subscript. The induced interpretation $[\![-]\!]_{\mathrm{Gl}_\lambda}$ extends the usual interpretation of terms in the syntactic category in the following sense: $[\![\Gamma \vdash t : \tau]\!]_{\mathrm{Gl}_\lambda}$ is a pair of the form $\left([\![\Gamma \vdash t : \tau]\!]_{\widehat{\mathrm{Ren}_\lambda}}, [\![\Gamma \vdash t : \tau]\!]\right)$ where $[\![-]\!]_{\widehat{\mathrm{Ren}_\lambda}} : \mathrm{Cn}_\lambda \to \widehat{\mathrm{Ren}_\lambda}$ is the functor induced by the $\widehat{\mathrm{Ren}_\lambda}$-lambda algebra over the family $\{\mathfrak{Ne}_\tau\}_{\tau \in \widetilde{T}}$ and the universal property of the classifying category.

Lemma 4.4.6. The composite $\pi_{\mathrm{sem}} \circ [\![-]\!]_{\mathrm{Gl}_\lambda} : \mathrm{Cn}_\lambda \to \mathrm{Cn}_\lambda$ is the identity endofunctor on Cn_λ.

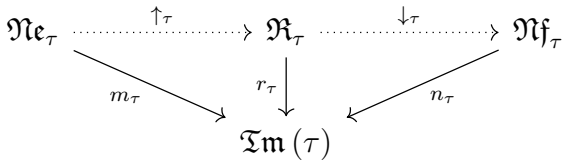
Proof. This follows from Theorem 1.5.11. In particular, since Cn_λ is the initial category with a model of the lambda calculus, we have that there is a unique functor $\mathrm{Cn}_\lambda \to \mathrm{Cn}_\lambda$ preserving the cartesian closed structure and the lambda algebra. The identity preserves the cartesian closed structure and the lambda algebra, so that uniqueness forces the desired equality. □

Now is a good time look ahead to where we're going, and see how what we've done so far will get us there.

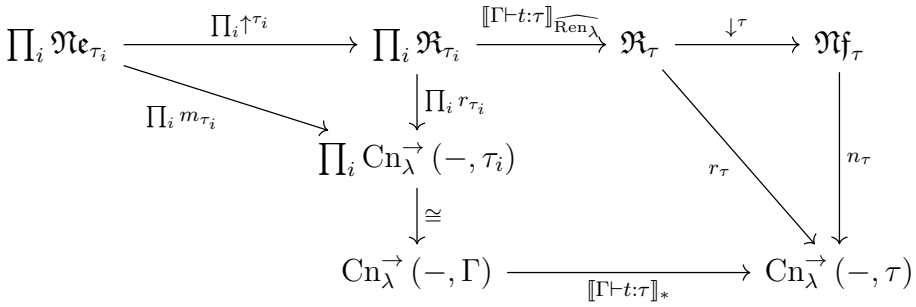
4.5 Taking stock: charting a path to normalization

We now have enough language to say exactly what it is we're hoping to acquire in this chapter. Let $\tau \in \widetilde{T}$ and suppose $[\![\tau]\!]_{\mathrm{Gl}_\lambda} = (\mathfrak{R}_\tau, q, \tau)$. We wish to come up with maps $\uparrow_\tau$ (pronounced "reflect", or for the LISPer "unquote") and $\downarrow_\tau$ (pronounced "reify", or for the LISPer "quote")[4] at each type $\tau \in \widetilde{T}$ to fill in the dotted arrows in the diagram

[4] In classical approaches to normalization by evaluation, reflection is understood as lifting syntax into a semantic domain: in practice, this looks like, say, taking syntactic functions in the object language to *actual functions* in the host language. Dually, reification is thought of as lowering semantic objects to a syntactic representation. This is why we use up and down arrow for these maps.



such that the diagram



commutes for $\Gamma = [x_1 : \tau_1, x_2 : \tau_2, \ldots, x_k : \tau_k]$.

Supposing we can achieve this, we have in particular that the evaluation

$$\left(n_\tau \circ \downarrow^\tau \circ [\![\Gamma \vdash t : \tau]\!]_{\widehat{\mathrm{Ren}_\lambda}} \circ \prod_{i \in \{1,\ldots,k\}} \uparrow^{\tau_i} \right) \left(\mathrm{var}_{\tau_i}(x_i) \right)_{i \in \{1,\ldots,k\}}$$

of the upper-right composite at a tuple of the variables contained in Γ has the same semantics as the input term t.

This is where our method shows its resemblance to the classical normalization by evaluation algorithm, in which the normalization function is given by *evaluating* the semantic interpretation of the input term on *lifted syntactic variables* before finally reifying the result, bringing it back down to syntax. Indeed, a normalization *algorithm* can be obtained from the above by stopping short of taking the semantic interpretation n_τ, as in:

$$\left(\downarrow^\tau \circ [\![\Gamma \vdash t : \tau]\!]_{\widehat{\mathrm{Ren}_\lambda}} \circ \prod_{i \in \{1,\ldots,k\}} \uparrow^{\tau_i} \right) \left(\mathrm{var}_{\tau_i}(x_i) \right)_{i \in \{1,\ldots,k\}}.$$

To come up with our desired reification and reflection maps, we shall produce, for each $\tau \in \widetilde{T}$, a pair of intermediate *glued* maps

$$\mu_\tau \xrightarrow{\Uparrow^\tau} [\![\tau]\!]_{\mathrm{Gl}_\lambda} \xrightarrow{\Downarrow^\tau} \eta_\tau$$

in the gluing category. Setting $\downarrow^\tau = \pi_{\mathrm{syn}}(\Downarrow^\tau)$ and $\uparrow^\tau = \pi_{\mathrm{syn}}(\Uparrow^\tau)$, we find that the triangles above follow from demonstrating that the above glued maps project in the semantics onto the identity: $\pi_{\mathrm{sem}}(\Downarrow^\tau) = \mathrm{id}_{[\![\tau]\!]} = \pi_{\mathrm{sem}}(\Uparrow^\tau)$. This follows by considering the universal property of a glued morphism. In the case of the reflect map, we get a diagram like this

$$\begin{array}{ccc} \mathfrak{Ne}_\tau & \xrightarrow{\pi_{\mathrm{syn}}(\Uparrow^\tau)=\uparrow^\tau} & \mathfrak{R}_\tau \\ \Big\downarrow{\scriptstyle m_\tau} & & \Big\downarrow{\scriptstyle r_\tau} \\ \mathfrak{Tm}\,(\tau) & \xrightarrow[\mathfrak{Tm}(\pi_{\mathrm{sem}}(\Uparrow^\tau))=\mathrm{id}]{} & \mathfrak{Tm}\,(\tau) \end{array}$$

which is the required triangle. In the reification case, we get the desired triangle along the same lines. Convinced thus, we can set about defining the required glued maps.

4.6 Glued reification and reflection

While defining these glued maps as pairs of suitable morphisms, we must take care to verify that the pairs form actual glued morphisms. Fortunately, the careful setup in the preceding pages makes this process almost automatic.

Definition 4.6.1. The glued maps are defined by induction on the type structure of $\widetilde{T}$:

- For a base type $\theta \in T$, we define $\Uparrow^\theta = \mathrm{id}_{\mu_\theta}$ and $\Downarrow^\theta = \mu_\theta \xrightarrow[\cong]{\mathrm{incl}_\theta} \eta_\theta$
- For the empty product/unit type $\mathbb{1}$, we define

  $$\Uparrow^{\mathbb{1}} = \left(\mu_{\mathbb{1}} \xrightarrow[\cong]{!} \mathbb{1}\right)$$

  and

  $$\Downarrow^{\mathbb{1}} = \left(\mathbb{1} \xrightarrow[\cong]{(\mathrm{unit},\mathrm{id})} \eta_{\mathbb{1}}\right)$$

  The former is the identity which is a map in the gluing category by definition. The latter is an already defined glued map from Definition 4.4.5.
- For types $\tau, \tau' \in \widetilde{T}$, we define

  $$\Uparrow^{\tau*\tau'} \colon \mu_{\tau*\tau'} \to [\![\tau]\!]_{\mathrm{Cn}_\lambda} \times [\![\tau']\!]_{\mathrm{Cn}_\lambda}$$

  as the pair of the following composites:

  $$\mu_{\tau*\tau'} \xrightarrow{\left(\mathrm{fst}^{\tau'}_{\tau},\pi_1\right)} \mu_\tau \xrightarrow{\Uparrow^\tau} [\![\tau]\!]_{\mathrm{Gl}_\lambda}$$
  $$\mu_{\tau*\tau'} \xrightarrow{\left(\mathrm{snd}^{\tau}_{\tau'},\pi_2\right)} \mu_{\tau'} \xrightarrow{\Uparrow^{\tau'}} [\![\tau']\!]_{\mathrm{Gl}_\lambda}$$

and define $\Downarrow^{\tau * \tau'} \colon [\![\tau]\!]_{\mathrm{Gl}_\lambda} \times [\![\tau']\!]_{\mathrm{Gl}_\lambda} \to \eta_{\tau * \tau'}$ as the composite

$$[\![\tau]\!]_{\mathrm{Gl}_\lambda} \times [\![\tau']\!]_{\mathrm{Gl}_\lambda} \xrightarrow{\Downarrow^\tau \times \Downarrow^{\tau'}} \eta_\tau \times \eta_{\tau'} \xrightarrow[\cong]{(\mathrm{pair}_{\tau * \tau'}, \mathrm{id})} \eta_{\tau * \tau'}.$$

In each case, the first pair in the composite is a glued map from Definition 4.4.5 while the second pair is a glued map by the induction hypothesis, so that the composite itself is a glued map. The product of these glued maps is a glued map by the universal property of products.

- For types $\tau, \tau' \in \widetilde{T}$, we define the reflection map

$$\Uparrow^{\tau \to \tau'} \colon \mu_{\tau \to \tau'} \to [\![\tau']\!]_{\mathrm{Gl}_\lambda}{}^{[\![\tau]\!]_{\mathrm{Gl}_\lambda}}$$

as the exponential transpose of the composite

$$\mu_{\tau \to \tau'} \times [\![\tau]\!]_{\mathrm{Gl}_\lambda} \xrightarrow{\mathrm{id} \times \Downarrow^\tau} \mu_{\tau \to \tau'} \times \eta_\tau \xrightarrow{(\mathrm{app}^\tau_{\tau'}, \epsilon)} \mu_{\tau'} \xrightarrow{\Uparrow^{\tau'}} [\![\tau']\!]_{\mathrm{Gl}_\lambda}.$$

By the universal property of the transpose, it suffices to show that the composite is a glued map of the appropriate domain and codomain. This follows from the induction hypothesis, the universal property of products (as in the previous induction step), and since $(\mathrm{app}^\tau_{\tau'}, \epsilon)$ is a glued map from Definition 4.4.5.

We define the reification map as

$$\Downarrow^{\tau \to \tau'} \colon [\![\tau']\!]_{\mathrm{Gl}_\lambda}{}^{[\![\tau]\!]_{\mathrm{Gl}_\lambda}} \to \eta_{\tau \to \tau'}$$

as the composite

$$[\![\tau']\!]_{\mathrm{Gl}_\lambda}{}^{[\![\tau]\!]_{\mathrm{Gl}_\lambda}} \xrightarrow{\left(\Downarrow^{\tau'}\right)^{\left(\Uparrow^\tau v_\tau\right)}} \eta_{\tau'}{}^{\nu_\tau} \xrightarrow[\cong]{(\mathrm{abs}_{\tau \to \tau'}, \mathrm{id})} \eta_{\tau \to \tau'}$$

where $v_\tau = (\mathrm{var}_\tau, \mathrm{id}) : \nu_\tau \to \mu_\tau$.

That this is a glued map follows immediately from Definition 4.4.5.

With the glued maps in hand, we can prove that they do nothing in the semantics. Naturally, the following theorem will prove crucial when demonstrating that our normalization function computes normal forms with the same semantics as the input term. This result is stated but not proven in Fiore's extended abstract [Fio02]. The proof is my own.

Theorem 4.6.2 (Reification and reflection are the identity in the semantics)**.** For each type $\tau \in \widetilde{T}$, we have the identities

$$\pi_{\mathrm{sem}}\left(\Uparrow^\tau\right) = \mathrm{id}_\tau = \pi_{\mathrm{sem}}\left(\Downarrow^\tau\right).$$

Proof. The proof proceeds, like the definitions of the reification/reflection maps, by induction on the structure of types:

- For a base type $\theta \in T$, the reflection map is $\Uparrow^{\theta} = \mathrm{id}_{\mu_\theta}$. By functoriality of the interpretation $[\![-]\!]_{\mathrm{Gl}_\lambda} : \mathrm{Cn}_\lambda \to \mathrm{Gl}_\lambda$ and Lemma 4.4.6, we have that $\pi_{\mathrm{sem}}\left(\mathrm{id}_{\mu_\theta}\right) = \mathrm{id}_{[\![\theta]\!]_{\mathrm{Cn}_\lambda}}$, as required. The reification map is $\Downarrow^{\theta} = \mu_\theta \xrightarrow[\cong]{\mathrm{incl}_\theta} \eta_\theta$ whose semantic component is the identity by Definition 4.4.5.

- For the empty product/unit type $\mathbb{1}$, the reflection map $\Uparrow^{\mathbb{1}}$ is the unique arrow from $\mu_{\mathbb{1}}$ into the terminal object of the gluing category. The interpretation $[\![-]\!]_{\mathrm{Gl}_\lambda}$ preserves the cartesian closed structure so that $\mu_\theta = \mathbb{1}$ and unique arrow in question is forced to be the identity in Gl_λ, whose semantic component is also the identity in Cn_λ.

  The reification map $\Downarrow^{\mathbb{1}}$ is the unit operation for the glued algebra and has the identity as its semantic component by definition.

- For $\tau, \tau' \in \widetilde{T}$, the reflection map $\Uparrow^{\tau\times\tau'}$ for their product is defined as the pair of the following composites:

$$\mu_{\tau*\tau'} \xrightarrow{\left(\mathrm{fst}^{\tau'}_{\tau},\pi_1\right)} \mu_\tau \xrightarrow{\Uparrow^{\tau}} [\![\tau]\!]_{\mathrm{Gl}_\lambda}$$
$$\mu_{tau*\tau'} \xrightarrow{\left(\mathrm{snd}^{\tau}_{\tau'},\pi_2\right)} \mu_{\tau'} \xrightarrow{\Uparrow^{\tau'}} [\![\tau']\!]_{\mathrm{Gl}_\lambda}$$

  so that

$$\begin{aligned}
\pi_{\mathrm{sem}}\left(\Uparrow^{\tau\times\tau'}\right) &= \left(\pi_{\mathrm{sem}}\left(\Uparrow^{\tau}\right) \circ \pi_1\right) \times \left(\pi_{\mathrm{sem}}\left(\Uparrow^{\tau'}\right) \circ \pi_2\right) \\
&= (\mathrm{id} \circ \pi_1) \times (\mathrm{id} \circ \pi_2) && \text{(induction hypothesis)} \\
&= \pi_1 \times \pi_2
\end{aligned}$$

  which is the identity on $[\![\tau]\!]_{\mathrm{Cn}_\lambda} \times [\![\tau']\!]_{\mathrm{Cn}_\lambda}$ as required.

  The reification map $\Downarrow^{\tau*\tau'}\colon [\![\tau]\!]_{\mathrm{Gl}_\lambda} \times [\![\tau']\!]_{\mathrm{Gl}_\lambda} \to \eta_{\tau*\tau'}$ is defined as the composite

$$[\![\tau]\!]_{\mathrm{Gl}_\lambda} \times [\![\tau']\!]_{\mathrm{Gl}_\lambda} \xrightarrow{\Downarrow^{\tau}\times\Downarrow^{\tau'}} \eta_\tau \times \eta_{\tau'} \xrightarrow[\cong]{(\mathrm{pair}_{\tau*\tau'},\mathrm{id})} \eta_{\tau*\tau'}$$

  whose semantic component is the identity by the induction hypothesis and the definition of the product functor in the gluing category.

- For $\tau, \tau' \in \widetilde{T}$, the reflection map $\Uparrow^{\tau\to\tau'}$ for the function type over these is defined as the exponential transpose of the composite

$$\mu_{\tau\to\tau'} \times [\![\tau]\!]_{\mathrm{Gl}_\lambda} \xrightarrow{\mathrm{id}\times\Downarrow^{\tau}} \mu_{\tau\to\tau'} \times \eta_\tau \xrightarrow{(\mathrm{app}^{\tau}_{\tau'},\epsilon)} \mu_{\tau'} \xrightarrow{\Uparrow^{\tau'}} [\![\tau']\!]_{\mathrm{Gl}_\lambda}$$

whence

$$\begin{aligned}
\pi_{\mathrm{sem}}\left(\Uparrow^{\tau\to\tau'}\right) &= \pi_{\mathrm{sem}}\left(\overbrace{\Uparrow^{\tau}\circ(\mathrm{app}^{\tau}_{\tau'},\epsilon)\circ(\mathrm{id}\times\Downarrow^{\tau})}\right)\\
&= \overbrace{\pi_{\mathrm{sem}}\left(\Uparrow^{\tau}\right)\circ\pi_{\mathrm{sem}}\left(\mathrm{app}^{\tau}_{\tau'},\epsilon\right)\circ\pi_{\mathrm{sem}}\left(\mathrm{id}\times\Downarrow^{\tau}\right)} && (2)\\
&= \widetilde{\mathrm{id}\circ\epsilon\circ(\mathrm{id}\times\mathrm{id})} && (3)\\
&= \widetilde{\mathrm{id}\circ\mathrm{id}} && (4)\\
&= \widetilde{\mathrm{id}}\\
&= \mathrm{id}
\end{aligned}$$

where the second equality follows from functoriality of π_{sem} and cartesian closure of π_{sem}; the third equality follows from the induction hypothesis and cartesian closure of π_{sem}; and the fourth equality follows from the universal property of the transpose, cartesian closure of π_{sem}, and because transposition fixes the identity.

Now the reification map $\Downarrow^{\tau\to\tau'}$ is defined as the composite

$$(\mathrm{abs}_{\tau\to\tau'},\mathrm{id})\circ\left(\Downarrow^{\tau'}\right)^{(\Uparrow^{\tau}v_{\tau})}$$

so that

$$\begin{aligned}
\pi_{\mathrm{sem}}\left(\Downarrow^{\tau\to\tau'}\right) &= \pi_{\mathrm{sem}}\left(\mathrm{abs}_{\tau\to\tau'},\mathrm{id}\right)\circ\left(\pi_{\mathrm{sem}}\Downarrow^{\tau'}\right)^{\pi_{\mathrm{sem}}(\Uparrow^{\tau})\circ\pi_{\mathrm{sem}}(v_{\tau})}\\
&= \mathrm{id}\circ(\mathrm{id})^{\mathrm{id}\circ\mathrm{id}} && \text{(induction hypothesis)}\\
&= \mathrm{id}^{\mathrm{id}}\\
&= \mathrm{id}^{*}\circ\mathrm{id}_{*}
\end{aligned}$$

which is the identity on the exponential $[\![\tau']\!]^{[\![\tau]\!]_{\mathrm{Cn}_\lambda}}_{\mathrm{Cn}_\lambda}$ in the classifying category.

□

4.7 A promise kept: a function from open terms to normal terms

Having defined our glued reify-reflect maps at each type, we have established the desired diagram from Section 4.5. We can now define a function $\mathrm{nf}^{\Gamma}_{\tau} : \mathfrak{L}_{\tau}(\Gamma) \to \mathfrak{Nf}_{\tau}(\Gamma)$ as the composite

$$\mathfrak{L}_{\tau}(\Gamma) \xrightarrow{l_{\tau}} \mathfrak{Tm}(\tau)(\Gamma) \xrightarrow{[\![-]\!]} \mathrm{Gl}_{\lambda}\left([\![\Gamma]\!],[\![\tau]\!]\right) \xrightarrow{(\Uparrow_{\Gamma}v_{\Gamma})^{*}\circ\Downarrow^{\tau}_{*}} \mathrm{Gl}_{\lambda}\left(\text{ﻉ}(\Gamma),\eta_{\tau}\right) \underset{\cong}{\xrightarrow{y}} \mathfrak{Nf}_{\tau}(\Gamma)$$

where

$$\begin{aligned} \Uparrow_\Gamma &= \prod_{(x:\tau)\in\Gamma} \Uparrow^\tau \\ v_\Gamma &= \overline{\text{よ}}\,(\Gamma) \xrightarrow{\cong} \prod_{(x:\tau)\in\Gamma} \nu_\tau \xrightarrow{\prod_{(x:\tau)\in\Gamma} v_\tau} \prod_{(x:\tau)\in\Gamma} \mu_\tau \\ y\,(d,\delta) &= d\,(\mathrm{id}_\Gamma) \end{aligned}$$

and recalling that $\nu_\tau \xrightarrow{v_\tau} \mu_\tau$ is the var_τ operation from the $\mathcal{NN}$-algebra on the gluing category.

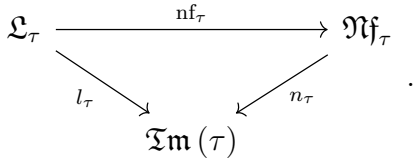
Since the final isomorphism in the composite is a form of the Yoneda lemma and hence given by evaluation at the identity, we have the following explicit formula for each term t:

$$\mathrm{nf}^\Gamma_\tau\,(t) = \left(\Downarrow^\tau [\![\Gamma \vdash t : \tau]\!] \Uparrow_\Gamma v_\Gamma\right)(\mathrm{id}_\Gamma)\,.$$

The function we've defined takes open terms to normal forms, but it remains to establish the various correctness properties characterizing a normalization function. The laborious setup work of this chapter turns out to allow us to do so with ease.

4.8 Reaping what we've sown: easy proofs of the correctness properties

Theorem 4.8.1 (Semantics preservation)**.** The following diagram commutes for every type $\tau \in \widetilde{T}$:



Proof. Per the version of the Yoneda lemma from Definition 4.4.1, we have that the composite $n_\tau \circ \left((d,\delta) \mapsto d\,(\mathrm{id})\right)$ of the semantic interpretation of normal forms in Cn_λ with the isomorphism witnessing the Yoneda lemma is the semantic projection π_{sem}. So it suffices to show that $\pi_{\mathrm{sem}}\left(\Downarrow^\tau [\![\Gamma \vdash t : \tau]\!]_{\mathrm{Gl}_\lambda} (\Uparrow_\Gamma v_\Gamma)\right) = [\![\Gamma \vdash t : \tau]\!]_{\mathrm{Cn}_\lambda}$. By Lemma 4.4.6, the interpretation of terms in the gluing category extends the interpretation of terms in the classifying category, so that $\pi_{\mathrm{sem}}\left([\![\Gamma \vdash t : \tau]\!]_{\mathrm{Gl}_\lambda}\right) = [\![\Gamma \vdash t : \tau]\!]_{\mathrm{Cn}_\lambda}$. Now the desired equality follows from observing that the glued morphisms $\Downarrow^\tau$, $\Uparrow^\tau$, and v_τ are all the identity in their semantic component, by Theorem 4.6.2 and Definition 4.4.5. □

Corollary 4.8.2. Because the interpretations l_τ and n_τ are just the semantic interpretation of terms, we have in particular for any term $\Gamma \vdash t : \tau$ that $\mathrm{nf}^\Gamma_\tau\,(t) \equiv_{\alpha\beta\eta} t$.

Theorem 4.8.3 (Equation preservation)**.** For every pair of terms $t, t' \in \mathfrak{L}_\tau\,(\Gamma)$, if $t \equiv_{\alpha\beta\eta} t'$ then $\mathrm{nf}^\Gamma_\tau\,(t) \equiv_{\alpha\beta\eta} \mathrm{nf}^\Gamma_\tau\,(t')$.

Proof. $\alpha\beta\eta$-equivalent terms have the same interpretation in the classifying category, because the classifying category is quotiented by definitional equality. Symbolically, we have $l_\tau(t) = l_\tau(t')$ from which the required equality is immediate by the definition of the normalization function as a composite starting with l_τ. □

Theorem 4.8.4 (Equation reflection)**.** For every pair of terms $t, t' \in \mathfrak{L}_\tau(\Gamma)$, if $\mathrm{nf}^\Gamma_\tau(t) \equiv_{\alpha\beta\eta} \mathrm{nf}^\Gamma_\tau(t')$ then $t \equiv_{\alpha\beta\eta} t'$.

Proof. The corollary to the semantics preservation theorem above together with transitivity of definitional equality give that

$$\begin{aligned} t &\equiv_{\alpha\beta\eta} \mathrm{nf}^\Gamma_\tau(t) \\ &\equiv_{\alpha\beta\eta} \mathrm{nf}^\Gamma_\tau\left(t'\right) && \text{(assumption)} \\ &\equiv_{\alpha\beta\eta} t' \end{aligned}$$

as required. □

Bibliography

[Ahm04] Amal Ahmed. *Semantics of Types for Mutable State*. Nov. 2004.

[Awo10] Steve Awodey. *Category theory*. 2nd ed. Oxford logic guides 52. Oxford ; New York: Oxford University Press, 2010. 311 pp. ISBN: 978-0-19-958736-0 978-0-19-923718-0.

[Bar12] Henk P. Barendregt. *The Lambda calculus its syntax and semantics*. Studies in Logic 40. London: College Publ, 2012. ISBN: 978-1-84890-066-0.

[BV14] Stephen D. Brown and Zvonko G. Vranesic. *Fundamentals of digital logic with Verilog design*. Third edition. New York: McGraw-Hill Higher Education, 2014. 847 pp. ISBN: 978-0-07-338054-4.

[BW] Michael Barr and Charles Wells. "TOPOSES, TRIPLES AND THEORIES". In: (), p. 302.

[BZ07] Nick Benton and Uri Zarfaty. "Formalizing and verifying semantic type soundness of a simple compiler". In: *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming - PPDP '07*. the 9th ACM SIGPLAN international conference. Wroclaw, Poland: ACM Press, 2007, p. 1. ISBN: 978-1-59593-769-8. DOI: 10.1145/1273920.1273922. URL: http://dl.acm.org/citation.cfm?doid=1273920.1273922 (visited on 12/01/2021).

[Chr] David Thrane Christiansen. *Checking Dependent Types with Normalization by Evaluation: A Tutorial*. URL: https://davidchristiansen.dk/tutorials/nbe/.

[Eat15] Ben Eater. *Making logic gates from transistors*. 2015. URL: https://www.youtube.com/watch?v=sTu3LwpF6XI.

[Fio02] Marcelo Fiore. "Semantic analysis of normalisation by evaluation for typed lambda calculus". In: *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming - PPDP '02*. the 4th ACM SIGPLAN international conference. Pittsburgh, PA, USA: ACM Press, 2002, pp. 26–37. ISBN: 978-1-58113-528-2. DOI: 10.1145/571157.571161. URL: http://portal.acm.org/citation.cfm?doid=571157.571161 (visited on 08/31/2021).

[Hara] Robert Harper. "How to (Re)Invent Tait' s Method". In: (), p. 6.

[Harb] Robert Harper. “Kripke-Style Logical Relations for Normalization”. In: (), p. 5.

[Har16] Robert Harper. *Practical foundations for programming languages*. Second edition. New York NY: Cambridge University Press, 2016. ISBN: 978-1-107-15030-0.

[Joh02] P. T. Johnstone. *Sketches of an elephant: a topos theory compendium*. Oxford logic guides. Oxford ; New York: Oxford University Press, 2002. ISBN: 978-0-19-852496-0 978-0-19-853425-9 978-0-19-851598-2.

[JT] Achim Jung and Jerzy Tiuryn. “A New Characterization of Lambda De nability”. In: (), p. 13.

[Law63] F. W. Lawvere. “FUNCTORIAL SEMANTICS OF ALGEBRAIC THEORIES”. In: *Proceedings of the National Academy of Sciences* 50.5 (Nov. 1, 1963), pp. 869–872. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.50.5.869. URL: http://www.pnas.org/cgi/doi/10.1073/pnas.50.5.869 (visited on 08/09/2021).

[LS89] Joachim Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics 7. Cambridge New York Port Chester [etc.]: Cambridge university press, 1989. ISBN: 978-0-521-35653-4.

[Mil78] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (Dec. 1978), pp. 348–375. ISSN: 00220000. DOI: 10.1016/0022-0000(78)90014-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0022000078900144 (visited on 12/01/2021).

[MM92] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: a first introduction to topos theory*. Universitext. New York: Springer-Verlag, 1992. 627 pp. ISBN: 978-0-387-97710-2 978-3-540-97710-0.

[Ngu] Quan Nguyen. “Incremental Improvements on the Placement and Routing of Minecraft Redstone Circuits”. In: (), p. 3.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. Cambridge, Mass: MIT Press, 2002. 623 pp. ISBN: 978-0-262-16209-8.

[Rie] Emily Riehl. “Category Theory in Context”. In: (), p. 258.

[SS18] Jonathan Sterling and Bas Spitters. “Normalization by gluing for free {\lambda}-theories”. In: *arXiv:1809.08646 [cs]* (Sept. 23, 2018). arXiv: 1809.08646. URL: http://arxiv.org/abs/1809.08646 (visited on 06/17/2021).

[Tay99] Paul Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, May 1, 1999.

[Wel09] Charles Wells. “Sketches: Outline with Reference”. Sept. 15, 2009.