Naïve logical relations in synthetic Tait computability[*]

JONATHAN STERLING

Aarhus University

Web: `http://www.jonmsterling.com`

Abstract

Logical relations are the main tool for proving **positive** properties of logics, type theories, and programming languages: canonicity, decidability, conservativity, computational adequacy, and more. Logical relations combine pure syntax with *non-syntactic* objects that are parameretized in syntax in a somewhat complex way; the sophistication of possible parameterizations makes logical relations a tool that is primarily accessible to specialists. In the spirit of Halmos' *Naïve Set Theory*, we advance a new viewpoint on logical relations based on **synthetic Tait computability**, an internal language for categories of logical relations. In synthetic Tait computability, logical relations are manipulated as if they were sets, making the essence of many complex logical relations arguments accessible to non-specialists.

Table of contents

*. This article has been written using GNU TEXMACS [Hoeven et al., 1998].

Acknowledgments. Thanks to Lars Birkedal for his comments on a draft of these lecture notes. I am grateful to Callen McGill for finding mistakes in an earlier draft. Finally, my sincere thanks to the denizens of the TEXMACS forum, who have helped me learn to use this excellent tool.

0 Meditations on synthetic methods

0.1 Synthesis, analysis, and the axiomatic method

The axiomatic method in mathematics is a tug of war between *analysis* and *synthesis*: analysis studies an object by modeling it with some substance that reliably behaves like the object in question, whereas synthesis takes stock of the competing models for an object and proceeds from an axiomatization of their essential properties. Analysis is important because it allows scientists to distinguish between empty fantasies and concepts that are grounded in reality by proceeding from the general to the particular; synthesis is no less important because it sharpens our view of the objects we study by summating our historical (analytical) experience in the form of axioms.

The canonical example of the synthetic method is the ancient Greek mathematician Euclid's plane geometry, which starts from an axiomatization of the properties of points, lines, circles, and angles as primitive notions. For instance:

1. A straight line can be drawn between any two points.
2. A line segment can be extended to a straight line.
3. There exists a circle with any center and radius.
4. All right angles are equal to each other.

The postulates above are descriptive of many **different** "models" for the concept of a line: this can be seen by considering the *parallel postulate*,[1] whose affirmation and negation are both consistent with the four postulates above. The revolution in modern mathematics brought on by the introduction of *analytic* (coordinate-based) methods under Descartes and Fermat should be understood as affirming the need for a plurality of models given that useful axiomatic systems can almost never uniquely determine their subject matter.

Analytic methods have been utterly dominant in recent mathematics: today, almost no mathematics is done purely synthetically, *i.e.* without reference whatsoever to a model. The modern viewpoint however accommodates (and greatly relies upon) the summation of analytic experience in the form of axioms, *e.g.* the axioms of set theory, the axioms of abelian categories [Grothendieck, 1957], or the axioms of homology theories [Eilenberg and Steenrod, 1945].

The use of axiomatic abstractions has many aspects in commonality with the classical synthetic method, but the attitude and orientation are different: today we use synthetic methods as a tool to manipulate and prove things about objects that are ultimately constituted in a "nuts and bolts" fashion. The benefit of a synthetic construction or argument is not only that it is intuitive and direct, but also that it applies all at once to many different possible models. Thus whereas in the past the synthetic and analytic viewpoints were opposed to each other, today they must be viewed as two parts of a whole whose interplay leads to new and useful developments in mathematics.

1. The statement of the parallel postulate: *If a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which the angles are less than two right angles.*

0.2 The relationship between internal and external language

The development of axiomatic set theory followed by the discovery of *sheaves* and *forcing* has led to a new style of synthetic mathematics that differs somewhat from Euclid's. In the new synthetic method, we start from a concrete domain concept (such as **topological spaces**, **smooth manifolds**, **computational datatypes**, **homotopy types**, *etc.*) and summon as if from the void a new kind of set theory in which instances of the original domain concept are treated *naïvely* as though they were just special sets.

The advantage of this approach is that we can completely do away with the complexities of checking the well-definedness (*e.g.* continuity, computability, *etc.*) of *functions* between domain objects, because these become just ordinary functions between sets. For instance, in ordinary topology we must be careful to ensure that functions are continuous; a special case of this arises in computer science, where we must constantly check that a function between *domains* (*e.g.* directed-complete partial orders, ω-cpos, Scott domains, *etc.*) preserves the appropriate directed suprema, an essentially bureaucratic task whose derilication can nonetheless lead to serious mistakes. Likewise in the context of theory of computation, we likewise have to check that a function between datatypes is tracked by a Turing machine.

In contrast, when (spaces, domains, datatypes, *etc.*) are viewed synthetically as if they were simply sets, any function between them will do. Obviously this sounds too good to be true, so there must be something that we have to give up in return for this simplicity; indeed, we must give up in many cases the unrestricted use of *classical logic*, in particular the law of the excluded middle and the axiom of choice. Thus the "set theory" generated by a given domain concept must be in general an **intuitionistic** set theory.

One of the consequences of passing to intuitionistic set theory is that we must deal with distinctions that did not have significance in the classical setting, such as the **failure of well-pointedness**. In classical set theory, well-pointedness means that a function $f : A \longrightarrow B$ is completely determined by its behavior on global elements $x \in A$ (*i.e.* functions $x : 1 \longrightarrow A$), whereas in intuitionistic set theory this need not hold.[2] In an intuitionistic setting, we must instead consider the composition of $f : A \longrightarrow B$ with arbitrary functions $a : I \longrightarrow A$ in order to fully characterize its behavior.[3] In such a setting, therefore, it is convenient to think of a function $a : I \longrightarrow A$ as a **generalized element** of A; then we are saying that $f : A \longrightarrow B$ is determined by its behavior on not only global elements but also generalized elements.

The new significance of generalized elements in intuitionistic set theories gives rise to a distinction between two kinds of language: **internal** and **external** language. External language is just ordinary mathematics, where we are explicit about the domains of generalized elements; external language thus allows us to distinguish between a global element and a parameterized element. In contrast, internal language expresses only that which applies to arbitrary generalized elements; internal statements can always be translated mechanically to external ones by explicitly reparameterizing all variable elements as generalized elements.[4] For example, even though the external statement of well-pointedness may not hold, the internal one does hold because its externalization is trivial:

> **Internal:** *For any functions* $f, g : A \longrightarrow B$*, if for all* $x : A$ *we have* $fx = gx$*, then* $f = g$*.*

2. Indeed, well-pointedness in this sense implies the law of the excluded middle.

3. In many cases, the domains I of the functions $a : I \longrightarrow A$ that we must compose with are drawn from a more restricted class (sometimes even a set) of distinguished sets.

4. The process of transforming internal statements to external ones is known as Kripke–Joyal semantics; this translation procedure is described in detail for first-order logic by Mac Lane and Moerdijk, 1992, and more recently for intuitionistic type theory by Awodey et al., 2021. In these lectures, we do not assume or require an understanding of the subtleties of these issues, but there is no escaping the fact that such a firm grasp of these matters will facilitate a deeper engagement with the material.

> **External:** *For any set I and functions $f, g : I \times A \longrightarrow B$, if for all $i : \mathcal{J} \longrightarrow I$ and $a : \mathcal{J} \longrightarrow A$ we have $f \circ \langle i, a \rangle = g \circ \langle i, a \rangle$, then $f = g$.*

Category theory is the language that is best equipped to express external statements and arguments, because category theory is explicit about the "domain of definition" of a given (generalized) element. In contrast, **type theory** is the language of internal mathematics; in type theory one works naïvely with "elements" $x : A$, and the constraints of the language ensure that everything you do is compatible with these elements being realized by generalized elements $x : I \longrightarrow A$.

Many aspects of synthetic mathematics are best realized in the internal / type theoretic manner, as it is natural for many objects to vary over a "domain of definition"; for instance, in computer science a program $\Gamma \vdash M : A$ can be thought of as a generalized element $M : \Gamma \longrightarrow A$, but it is often useful to speak simply of elements of type A when the observations we want to make are closed under substitution.

0.3 Using synthetic methods to prove analytical facts

Today when synthetic methods are employed, it is usually with the intention of proving something about a concrete (analytically constituted) object. For instance, synthetic differential geometry [Kock, 2006] is used to prove things about actual smooth manifolds; synthetic domain theory can be used to prove things about actual ω-cpos [Fiore and Plotkin, 1996]. In these lectures, we will use **synthetic Tait computability** to prove things about the *actual* syntax of actual programming languages!

In order to substantiate the link between the synthetic world and the analytic world, it is necessary to construct an actual analytical model of the synthetic theory at hand and to understand the way that synthetic statements get unravelled to analytical statements. Our approach (which is by no means the only valid approach) is to use internal language when working synthetically, and only use external language at the boundary between the synthetic theory and its analytic model.

It cannot be escaped that it requires some expertise to verify the correct relationship between synthetic and analytic notions; what the synthetic viewpoint offers, however, is to *isolate* the need for this expertise to the margins of a proof and allow as much as possible of the main argument to follow in a naïve way.

0.4 Who is this for?

These lectures aim to be accessible to those who have some experience *using* dependent type theory in proof assistants like Coq or Agda; such a reader will be comfortable with concepts like dependent records, dependent functions, equality types, and type universes. We do not assume knowledge of category theory, though it will come up in a few places.

1 Synthetic canonicity for typed λ-calculus

We begin our invitation to synthetic methods with a particularly simple example that nonetheless illustrates two important elements of our toolkit: the naïve manipulation of syntactic terms as if they were ordinary functions, and the construction of synthetic logical relations over this syntax through a *phase distinction* between "object-space" and "meta-space".

1.1 What is canonicity?

Typed λ-calculus can be viewed in two ways: as a programming language, or as the *internal language* of cartesian closed categories. The former viewpoint is meaningful when λ-calculus is understood to include a type of *observations* obs with two distinct elements accept and reject; then any function $\sigma \longrightarrow \tau$ in typed λ-calculus can be thought of as a program by studying its composition with functions (observations) of type $\tau \longrightarrow \mathsf{obs}$. What makes this analogy between λ-terms and programs tenable is the **canonicity** property which characterizes the closed elements of type obs, *i.e.* the elements that have no free variables:

Canonicity. The only **closed** elements of type obs are accept and reject.

Intuitively the canonicity property states that regardless of all the other features of the language (*e.g.* function types, product types, *etc.*), the elements of type obs can be thought of as computing states of an abstract machine. Indeed, because the derivations of a formalism for typed λ-calculus are recursively enumerable, the canonicity property implies that there exists a mechanical procedure to assign acccept/reject states to programs of type obs.[5]

Canonicity is not the most important metatheoretic property of a language, but it serves as a good introduction to the study of such properties because it is (1) not difficult to prove, and (2) introduces important concepts such as the distinction between an *element* and a *closed element*. Canonicity is usually proved using logical relations or *Tait's method*; in this lecture, we will show how to use a synthetic version of logical relations to prove canonicity. We have named our methodology ***synthetic Tait computability*** [Sterling, 2021].

1.2 Tagless encoding of typed λ-calculus

For us, the typed λ-calculus is an *equational theory*: there is a sort of types, and for each type there is a sort of elements of that type. Computational principles like β-reduction and η-expansion are encoded as directionless equations. Thus the structure of typed λ-calculus can be encoded in type theory as a *dependent record* or *module signature*. Here is the beginning of our signature:

begin signature $\mathsf{STLC}_{\mathcal{U}}$
  $\mathsf{type} : \mathcal{U}$
  $\mathsf{el} : \mathsf{type} \longrightarrow \mathcal{U}$

Above we have parameterized the signature in a universe $\mathcal{U}$ in which the *sorts* of the λ-calculus are taken; this will be useful later on, but feel free to ignore it. So far we have defined the sort of types and the *dependent* sort of elements of a given type. We first close our calculus under function types; this is done by adding a constant to type for the function space, and then adding abstraction and application operators to el.

$$\mathsf{arr} : \mathsf{type} \longrightarrow \mathsf{type} \longrightarrow \mathsf{type}$$
$$\mathsf{lam} : (\sigma, \tau : \mathsf{type}) \longrightarrow (\mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\tau) \longrightarrow \mathsf{el}\,(\mathsf{arr}\,\sigma\,\tau)$$
$$\mathsf{app} : (\sigma, \tau : \mathsf{type}) \longrightarrow \mathsf{el}\,(\mathsf{arr}\,\sigma\,\tau) \longrightarrow \mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\tau$$

Remark 1. Note how we have used the ambient function space to represent the *binder* in the lam constructor above. This is often called *higher-order abstract syntax*, and is characteristic of the *finally tagless* encodings [Carette et al., 2007]. The use of the function space for binding is very important: it expresses the sense in which our treatment of syntax is "naïve": intuitively, a binder ought to be a function of all the things that can be substituted into it — a principle known as the Yoneda lemma in category theory, which Hofmann has used to deliver a mathematical justification for the naïve view of binders as functions [Hofmann, 1999].

5. Note that this procedure takes the form of unbounded search rather than operational reduction.

What about computation rules? We can use *equality types* to specify these as follows:

$$\mathsf{arr}_\beta : (\sigma, \tau : \mathsf{type})\,(u : \mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\tau) \longrightarrow \mathsf{app}\,\sigma\,\tau\,(\mathsf{lam}\,\sigma\,\tau\,u)\,v = u\,v$$
$$\mathsf{arr}_\eta : (\sigma, \tau : \mathsf{type})\,(u : \mathsf{el}\,(\mathsf{arr}\,\sigma\,\tau)) \longrightarrow u = \mathsf{lam}\,\sigma\,\tau\,(\lambda x.\mathsf{app}\,\sigma\,\tau\,u\,x)$$

Convention 2. In both informal mathematics *and* mechanized mathematics, one has access to notational conveniences such as *implicit arguments*. The use of implicit arguments does not change the meaning of an expression, but it places an additional burden on its reader to fill in potentially ambiguous information that has been omitted; in the case of a tool like Agda or Coq or Twelf, the "reader" is a piece of software, and in the case of the present paper, the reader is *You*! We will adopt implicit arguments in our notations to make it easier to see the important parts of expressions. Under the implicit notation, what we have written so far looks like this:

begin signature $\mathsf{STLC}_{\mathcal{U}}$
$$\mathsf{type} : \mathcal{U}$$
$$\mathsf{el} : \mathsf{type} \longrightarrow \mathcal{U}$$
$$\mathsf{arr} : \mathsf{type} \longrightarrow \mathsf{type} \longrightarrow \mathsf{type}$$
$$\mathsf{lam} : (\mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\tau) \longrightarrow \mathsf{el}\,(\mathsf{arr}\,\sigma\,\tau)$$
$$\mathsf{app} : \mathsf{el}\,(\mathsf{arr}\,\sigma\,\tau) \longrightarrow \mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\tau$$
$$\mathsf{arr}_\beta : \mathsf{app}\,\sigma\,\tau\,(\mathsf{lam}\,\sigma\,\tau\,u)\,v = u\,v$$
$$\mathsf{arr}_\eta : (u : \mathsf{el}\,(\mathsf{arr}\,\sigma\,\tau)) \longrightarrow u = \mathsf{lam}\,\sigma\,\tau\,(\lambda x.\mathsf{app}\,\sigma\,\tau\,u\,x)$$

Next we add binary product types in the same way as above: we add a type constant, some term constants, and some equations.

$$\mathsf{prod} : \mathsf{type} \longrightarrow \mathsf{type} \longrightarrow \mathsf{type}$$
$$\mathsf{pair} : \mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\tau \longrightarrow \mathsf{el}\,(\mathsf{prod}\,\sigma\,\tau)$$
$$\mathsf{fst} : \mathsf{el}\,(\mathsf{prod}\,\sigma\,\tau) \longrightarrow \mathsf{el}\,\sigma$$
$$\mathsf{snd} : \mathsf{el}\,(\mathsf{prod}\,\sigma\,\tau) \longrightarrow \mathsf{el}\,\tau$$
$$\mathsf{prod}_{\beta 1} : \mathsf{fst}\,(\mathsf{pair}\,u\,v) = u$$
$$\mathsf{prod}_{\beta 2} : \mathsf{snd}\,(\mathsf{pair}\,u\,v) = v$$
$$\mathsf{prod}_\eta : (u : \mathsf{el}\,(\mathsf{prod}\,\sigma\,\tau)) \longrightarrow u = \mathsf{pair}\,(\mathsf{fst}\,u)\,(\mathsf{snd}\,u)$$

Exercise 1. Write out for yourself a fully explicit version of the signature fragment above, with all implicit arguments replaced by explicit arguments.

Exercise 2. Construct an element of type $\mathsf{arr}\,(\mathsf{prod}\,\sigma\,\tau)\,(\mathsf{prod}\,\tau\,\sigma)$.

Finally we add a base type; to balance familiarity with simplicity, we choose the *booleans*. These will serve as our type of *observations* relative to which canonicity is stated (recall Section 1.1).

$$\mathsf{bool} : \mathsf{type}$$
$$\mathsf{true} : \mathsf{el}\,\mathsf{bool}$$
$$\mathsf{false} : \mathsf{el}\,\mathsf{bool}$$
$$\mathsf{case} : \mathsf{el}\,\mathsf{bool} \longrightarrow \mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\sigma \longrightarrow \mathsf{el}\,\sigma$$
$$\mathsf{bool}_{\beta 1} : \mathsf{case}\,\mathsf{true}\,u\,v = u$$
$$\mathsf{bool}_{\beta 2} : \mathsf{case}\,\mathsf{false}\,u\,v = v$$

Is there an η-law? This is a point on which different researchers may disagree at different times, depending on what their goals are. We choose the add the η-law form uniformity's sake:

$$\mathsf{bool}_\eta : (u : \mathsf{el}\,\mathsf{bool} \longrightarrow \mathsf{el}\,\sigma) \longrightarrow u = \lambda x.\,\mathsf{case}\,x\,(u\,\mathsf{true})\,(u\,\mathsf{false})$$

And we are done!

end signature $\mathsf{STLC}_{\mathcal{U}}$

1.3 Example: a structure for the typed λ-calculus

In this section, we will explore what it means to construct a *structure* for the typed λ-calculus, *i.e.* an element of the dependent record type STLC that we have defined in Section 1.2. For our example, we will define the "standard" model in which the sort of types is interpreted by a universe. To that end, let $\mathcal{U} : \mathcal{V}$ be a pair of fixed universes.

begin structure Simple : $\mathsf{STLC}_{\mathcal{V}}$

To define the constant $\mathsf{type} : \mathcal{V}$ we will use the lower universe $\mathcal{U}$. The constant el therefore has type $\mathcal{U} \longrightarrow \mathcal{V}$; we will use the (implicit) coercion from the lower universe to the upper universe:

$$\mathsf{type} := \mathcal{U}$$
$$\mathsf{el}\ \sigma := \sigma$$

Function types are interpreted as function types!

$$\mathsf{arr}\ \sigma\ \tau := \sigma \longrightarrow \tau$$
$$\mathsf{lam}\ u := u$$
$$\mathsf{app}\ u\ v := u\,v$$

When we interpret equational constants like arr_β, we must check that the desired equation actually holds; for example, we need to check that $\mathsf{app}\,(\mathsf{lam}\ u)\ v = u\,v$, but this holds immediately by unfolding the definitions we have given above. We will write $\star$ for the unique element of any true equality type:

$$\mathsf{arr}_\beta := \star$$
$$\mathsf{arr}_\eta := \star$$

The interpretation of product types works the same way:

$$\mathsf{prod}\ \sigma\ \tau := \sigma \times \tau$$
$$\mathsf{pair}\ u\ v := (u, v)$$
$$\mathsf{fst}\ (u, v) := u$$
$$\mathsf{snd}\ (u, v) := v$$
$$\mathsf{prod}_{\beta 1} := \star$$
$$\mathsf{prod}_{\beta 2} := \star$$
$$\mathsf{prod}_\eta\ u := \star$$

We interpret the booleans using a coproduct:

$$\mathsf{bool} := \mathbf{1} + \mathbf{1}$$
$$\mathsf{true} := \mathsf{inl}\ \star$$
$$\mathsf{false} := \mathsf{inr}\ \star$$

The case expression is defined by pattern matching on the coproduct:

$$\mathsf{case}\ (\mathsf{inl}\ \star)\ u\ v := u$$
$$\mathsf{case}\ (\mathsf{inr}\ \star)\ u\ v := v$$

$\mathsf{bool}_{\beta 1} := \star$
$\mathsf{bool}_{\beta 2} := \star$
$\mathsf{bool}_{\eta}\, u := \star$
end structure Simple

1.4 Towards canonicity: a phase distinction for object and meta

Recall the informal statement of canonicity for typed λ-calculus:

> **Canonicity.** There are exactly two **closed** elements of type bool, namely true and false.

We do not actually have a rich enough vocabulary at the moment to make such a statement precise in the synthetic world relative to our signature STLC. First of all, the notion of an "element of type bool" is meaningful only with respect to a specific structure for STLC; moreover, we do not have a precise notion of **closed** element in our synthetic setting. The solution to these problems is to *postulate* out of thin air a **phase distinction** in our ambient type theory between **object-space** and **meta-space** constructs.

When we refer to something as "object-space" we mean to abstract the way that some objects can be built up by gluing pieces of actual object-syntax together. For instance, the collection type of all object-language types is object-space by definition, as it corresponds to an object-judgment; but it is *also* useful to be able to refer to things like the coproduct type + type as object-space even if such a coproduct does not appear in the judgmental structure of typed λ-calculus. In contrast, "meta-space" notions arise when we consider things that cannot be expressed by gluing together object-language constructs, such as the concept of a **closed element**.

Every type in the ambient language will have both an object-space and a meta-space part, but in some cases one of these components may be "mute" in a certain sense that will be made precise. A type whose object-space component is mute will be thought of as an ordinary set, *e.g.* the set of closed elements of a given object-type.

Warning 3. So far we have been investigating the encoding of an "object language" within an ambient intuitionistic synthetic type/set theory with constructs for defining signatures and structures. It is important to avoid confusing the notion of "meta" discussed in this section with the "ambient" language; we intend to formalize the relationship between "object" and "meta" within this ambient synthetic language.

To introduce a new **phase distinction**, we assume an indeterminate *proposition* in the ambient synthetic language.[6]

Postulate. (Phase distinction) We assume a proposition $\mathbf{\Phi}$, *i.e.* a type whose elements are all equal.

Remember that our ambient theory is intuitionistic, so it need not be the case that $(\mathbf{\Phi} = \top) \vee (\mathbf{\Phi} = \bot)$. Despite the seeming triviality of such a postulate, its assumption generates a great deal of rich structure that we shall use to access important notions like *syntax* and *closed elements* from the synthetic world.

Definition 4. *An object A is called $\mathbf{\Phi}$-**transparent** when the constant function $(\lambda x.\lambda z.x) : A \longrightarrow (\mathbf{\Phi} \longrightarrow A)$ is an isomorphism. Unfolding definitions, this means more explicitly the following*

6. A proposition is defined to be a type whose elements, if they exist, are all equal to each other. When a proposition has an element, it is referred to as "true". We further assume that any two true propositions are equal.

For any function $u : \Phi \longrightarrow A$ there exists a unique *element $a : A$ such that $u = \lambda z.\, a$.*

Definition 5. *An object A is called **Φ-sealed** when the projection function $\pi_1 : \Phi \times A \longrightarrow \Phi$ is an isomorphism. This can be equivalently restated as the condition that A has exactly one element under the assumption of $\Phi = \top$.*

The notions of transparency and sealing described above make sense for any proposition; but in the specific case of Φ, we impose the following terminology to make our intuitions explicit:

1. A Φ-transparent object is called ***object-space***.
2. A Φ-sealed object is called ***meta-space***.

The proposition Φ can be thought of as representing a "region" of space where only object-space stuff can be seen; we say that we are ***"in object-space"*** when Φ is true. One should think of Φ-transparent objects as those that come from object-space; so if you are trying to construct an element of an object-space type A, you may assume without loss of generality that $\Phi = \top$. On the other hand, a meta-space object is one that has no extent within object-space: thus for a Φ-sealed type A, we cannot extract any information from an element of A when $\Phi = \top$, since A becomes a singleton within the object-space.

Remark 6. It is also possible to speak of "meta-space", but unlike object-space it is not parameterized by a proposition (*e.g.* we cannot say that one is "in meta-space" when $\neg\Phi$ is true). This asymmetry corresponds to the topological fact that the complement of an open subspace is not necessarily an open subspace. Nonetheless, we can say that an object comes "from" meta-space when it is Φ-sealed.

1.4.1 Projecting the object-space component of a type

Let A be an arbitrary type; the ***object-space component*** of A is defined to be the function space $\Phi \longrightarrow A$. Intuitively, this definition comes from the fact that assuming $\Phi = \top$ causes you to "enter object-space", which causes the meta-space components of A to evaporate.

Exercise 3. Check that for any A, the object-space component of $\Phi \longrightarrow A$ is in fact object-space / Φ-transparent in the sence of Definition 4.

How do we know that we have correctly defined the object-space component? Because we can verify the following universal property:

Lemma 7. *Let A be an arbitrary type and let B be an object-space type; then the function $\lambda f.\, \lambda a.\, f(\lambda z.\, a) : ((\Phi \longrightarrow A) \longrightarrow B) \longrightarrow (A \longrightarrow B)$ is an isomorphism.*

Because Φ is a proposition, we know that any two elements $u, v : \Phi$ are interchangeable; this irrelevance suggests a more expedient notation for working with the object-space component.

Notation 8. *We will write $\Phi \Longrightarrow A$ to for the object-space component of A, but we will leave λ-abstraction and application for this particular function space implicit. In particular, when $u : A$ we will also write $u : \Phi \Longrightarrow A$ to mean $\lambda z.u$. This abuse of notation causes no ambiguity, and leads to much easier to read expressions.*

It often happens that we have a type $A : \Phi \Longrightarrow \mathcal{U}$, *i.e.* a type in object-space, and we want to extend it out of object-space into ordinary space as an element of $\mathcal{U}$. This is achieved by taking the *dependent product* $(z : \Phi) \longrightarrow A\, z$. We extend our implicit notation to handle the dependent case as follows: when $A : \Phi \Longrightarrow \mathcal{U}$, we will write $\Phi \Longrightarrow A$ for the dependent product $(z : \Phi) \longrightarrow A$, with likewise implicit λ-abstraction and application.

1.4.2 Projecting the meta-space component of a type

In contrast, the ***meta-space component*** of A is a bit harder to define. We know that it needs to be $\mathbf{\Phi}$-sealed, so it must have at least a function from $\mathbf{\Phi}$. Because it should be possible to project from an element of type A is meta-space component, we also need a function from A. Moreover, because the meta-space component must contract to a singleton within object-space, we need the image of A in the meta-space component to be equal to the image of $\mathbf{\Phi}$! This suggests the following definition of the meta-space component $\mathbf{\Phi} \bullet A$ as a quotient of a coproduct:

$$\mathbf{\Phi} \bullet A := (\mathbf{\Phi} + A)/{\sim} \qquad u \sim v \Longleftrightarrow \mathbf{\Phi} \vee u = v \qquad \text{(the meta-space component)}$$

We will write $\eta_{\mathbf{\Phi}\bullet} : A \to \mathbf{\Phi} \bullet A$ for the function that sends $a : A$ to the equivalence class $[\mathrm{inr}\, a]_\sim$.

Remark 9. For those who are familiar with some category theory, the meta-space component $\mathbf{\Phi} \bullet A$ is the *pushout* of the two product projections $\mathbf{\Phi} \leftarrow \mathbf{\Phi} \times A \to A$.

$$\begin{array}{ccc} \mathbf{\Phi} \times A & \xrightarrow{\pi_2} & A \\ {\scriptstyle \pi_1}\downarrow & \text{(cocart.)} & \downarrow{\scriptstyle \eta_{\mathbf{\Phi}\bullet}} \\ \mathbf{\Phi} & \xrightarrow[\star]{} & \mathbf{\Phi} \bullet A \end{array}$$

In these lectures we are careful not to require knowledge of category theory or pushouts, but it cannot be denied that knowledge of categorical language confers a definite advantage.

Exercise 4. Verify that for any type A, the meta-space component $\mathbf{\Phi} \bullet A$ is $\mathbf{\Phi}$-sealed/meta-space in the sense of Definition 5.

Exercise 5. Show that the metacomponent $\mathbf{\Phi} \bullet \phi$ of any proposition ϕ is again a proposition, namely the disjunction $\mathbf{\Phi} \vee \phi$.

We can justify the definition of the meta-space component by proving its universal property:

Lemma 10. *Let A be an arbitrary type and let B be a meta-space type; then the function $\lambda f.\lambda a.f(\eta_{\mathbf{\Phi}\bullet} a) : (\mathbf{\Phi} \bullet A \to B) \to (A \to B)$ is an isomorphism.*

Warning 11. Although we have $(\mathbf{\Phi} \Rightarrow \mathbf{\Phi} \bullet A) \cong \mathbf{1}$ for any A, the same does not hold for the meta-space component $\mathbf{\Phi} \bullet (\mathbf{\Phi} \Rightarrow A)$ which can be non-trivial in general. In fact, in Section 1.4.3 we will use the meta-space component of an object-space type A as a synthetic analogue to the "set of closed elements of A".

The meta-space component operation $\mathbf{\Phi} \bullet -$ is a monad; for instance, we can define the Kleisli extension $\{x \leftarrow u; bx\}$ of a function $b : A \to \mathbf{\Phi} \bullet B$ applied to an element $u : \mathbf{\Phi} \bullet A$ by pattern matching in the following way:

$$\begin{aligned} \{x \leftarrow \eta_{\mathbf{\Phi}\bullet} u; bx\} &:= bu \\ \{x \leftarrow \star; bx\} &:= \star \end{aligned}$$

In fact, the same notation may be applied in a more general scenario. Suppose that $B : \mathbf{\Phi} \bullet A \to \mathcal{U}$ is a family of $\mathbf{\Phi}$-sealed types, *i.e.* each Bu is $\mathbf{\Phi}$-sealed. Then we will write $\{x \leftarrow u; bx\} : Bu$ for the element defined by the same clauses as above.

Construction 12. (Delayed substitutions) In addition to being an ordinary monad, $\mathbf{\Phi} \bullet -$ also supports a related construct on the *type-level* that may be less familiar. Suppose that $B : A \longrightarrow \mathcal{U}$ is a family of types such that each Bu is $\mathbf{\Phi}$-sealed; then we may extend B to a family of types $\mathbf{\Phi} \bullet A \longrightarrow \mathcal{U}$ that takes $u : \mathbf{\Phi} \bullet A$ to a $\mathbf{\Phi}$-sealed type written $\mathbf{\Phi} \bullet [x \longleftarrow u]. Bx$. Up to isomorphism, this type could be defined as follows:

$$(\mathbf{\Phi} \bullet [x \longleftarrow u]. Bx) \cong (x : \eta^{-1}_{\mathbf{\Phi}\bullet} u) \longrightarrow Bx$$

For convenience, we assert that $(\mathbf{\Phi} \bullet [x \longleftarrow \eta_{\mathbf{\Phi}\bullet} v]. Bx) = \mathbf{\Phi} \bullet Bv$ strictly. To introduce an element of $\mathbf{\Phi} \bullet [x \longleftarrow u]. Bx$ given $b : (x : \eta^{-1}_{\mathbf{\Phi}\bullet} u) \longrightarrow Bx$, we simply write $\{x \longleftarrow u; bx\}$. Finally we extend all these notations over arbitrary telescopes $\Psi \equiv (x : A, y : Bx, \dots)$, *etc.* Some readers may recognize the ***delayed substitutions*** of guarded dependent type theory (Bizjak et al., 2016) in our notation.

1.4.3 The generic syntactic structure

We expressed our *intention* that object-space be populated by things that can be built up from object-space constructs. We begin to substantiate that intuition by **postulating** a *generic* structure for the typed λ-calculus.

Postulate. (The generic structure) We assume an object-space structure $\mathsf{M} : \mathbf{\Phi} \Longrightarrow \mathsf{STLC}_{\mathcal{U}}$.

The postulated structure M represents the *generic model* of typed λ-calculus from within the synthetic setting; the idea is that if we prove something about M in the synthetic world, then some corresponding statement will hold of the actual syntax of typed λ-calculus outside the synthetic world.

1.4.4 The generic closed instance of a synthetic element

To express the canonicity property, we will need a way to talk about **closed elements**! In the synthetic setting, we will access closed elements simply by projecting the meta-space component of the collection of object-elements. Given a object-space type $A : \mathbf{\Phi} \Longrightarrow \mathcal{U}$, the meta-space type of ***closed elements*** of A is simply defined to be the meta-space component $\mathbf{\Phi} \bullet (\mathbf{\Phi} \Longrightarrow A)$.

The correct intuition for closed elements in the synthetic setting is a little subtle. In the analytic account of syntax, an element is either open or closed by definition depending on what context it is constructed in. In contrast, the synthetic setting is oblivious to contexts and so when we have a synthetic element $u : \mathbf{\Phi} \Longrightarrow \mathsf{M.el}\, \sigma$, we ought to think of it as corresponding to an open term in an ambient context determined by the environment; on the other hand, an element of the meta-space component $\mathbf{\Phi} \bullet (\mathbf{\Phi} \Longrightarrow \mathsf{M.el}\, \sigma)$ should be thought of as standing not for a *single* closed element of type σ but rather for *all* the closed instances of some (unspecified) open element.

Thus given an element $u : \mathbf{\Phi} \Longrightarrow \mathsf{M.el}\, \sigma$, the meta-space component $\eta_{\mathbf{\Phi}\bullet} u : \mathbf{\Phi} \bullet (\mathbf{\Phi} \Longrightarrow \mathsf{M.el}\, \sigma)$ can be thought of as the ***generic closed instance*** of u; intuitively, anything proved about $\eta_{\mathbf{\Phi}\bullet} u$ will ultimately hold (externally) of *all* closed instance of the open term u. The existence of magical elements that stand in for a whole class of elements is one of the strengths of the synthetic approach.[7]

7. A similar phenomenon is observed in the history of algebraic geometry; the Italian geometers of the early 20th century worked intuitively with a quite unrigorous notion of the "generic point on a curve" — a magical point that somehow concentrates the properties of all the other points. It was not until the introduction of scheme theory that the geometrical intuition of the generic point was fully substantiated by new language in which it arises naturally and rigorously. We view the idea of the ***generic closed instance*** in the synthetic study of syntax as an analogous phenomenon, in which new geometrical intuitions are substantiated by the introduction of new language.

Remark 13. Note that we have not made any assumptions yet that guarantee any relationship between "synthetic closed elements" and actual closed terms in a formal presentation of typed λ-calculus. Such a relationship will be established when we relate our synthetic world with the external world.

1.5 A synthetic logical predicate for canonicity

A *synthetic logical predicate* on the generic structure $\mathsf{M} : \mathbf{\Phi} \Longrightarrow \mathsf{STLC}_{\mathcal{U}}$ is nothing more than another structure $\mathsf{M}^* : \mathsf{STLC}_{\mathcal{V}}$ whose object-space component is M modulo the inclusion $\mathcal{U} \subseteq \mathcal{V}$. The idea is that the meta-space component of M^* will carry the proofs that each closed element of type bool is either true or false and that this property is preserved by all higher type structure.

1.5.1 Extension types

In order to express the sense in which the M^* restricts to M in object-space, we introduce a new type constructor called the *extension type*; our extension types are a special case of those of Riehl and Shulman, 2017, but we do not require knowledge of the general notion.

Definition 14. *Given a type A and an object-space element $a : \mathbf{\Phi} \Longrightarrow A$, the **extension type** of a is defined to be the type $\{A \mid \mathbf{\Phi} \hookrightarrow a\}$ of all elements of A that restrict within object-space to a; formally we can define $\{A \mid \mathbf{\Phi} \hookrightarrow a\}$ as a subset type:*

$$\{A \mid \mathbf{\Phi} \hookrightarrow a\} := \{x : A \mid \mathbf{\Phi} \Longrightarrow x = a\}$$

The following observation is immediate, but nonetheless very important.

Lemma 15. *Every extension type $\{A \mid \mathbf{\Phi} \hookrightarrow a\}$ is $\mathbf{\Phi}$-sealed.*

Proof. It suffices to check that when $\mathbf{\Phi} = \top$, the type $\{A \mid \mathbf{\Phi} \hookrightarrow a\}$ has exactly one element. Because we are "in object-space" by virtue of our assumption, we have $a : A$ which we now claim to be the only element of $\{A \mid \mathbf{\Phi} \hookrightarrow a\}$. Fix any $b : \{A \mid \mathbf{\Phi} \hookrightarrow a\}$, *i.e.* any $b : A$ such that $\mathbf{\Phi} \Longrightarrow b = a$; our assumption implies that $b = a$. □

Now we may express the type of the synthetic logical predicate more precisely:

$$\mathsf{M}^* : \{\mathsf{STLC}_{\mathcal{V}} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M}\}$$

1.5.2 Refinement types

It is well and good to be able to *state* the extension type $\{\mathsf{STLC}_{\mathcal{V}} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M}\}$, but how can we construct an element of this type? Such an element must have, in particular, a field $\mathsf{M}^*.\mathsf{type} : \{\mathcal{V} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M}.\mathsf{type}\}$; to decide how we should define $\mathsf{M}^*.\mathsf{type}$, we recall that in an ordinary logical predicates model, the sort of types is interpreted by the collection of object-space types equipped with predicates on their closed terms. In the synthetic setting, there is a more direct way to achieve roughly the same thing: we want an element of $\mathsf{M}^*.\mathsf{type}$ to be given by an element $\sigma : \mathbf{\Phi} \Longrightarrow \mathsf{M}.\mathsf{type}$ together with a type $\sigma' : \mathcal{U}$ that restricts in object-space to $\mathsf{M}.\mathsf{el}\,\sigma$. In other words, we would like to define $\mathsf{M}^*.\mathsf{type}$ as follows the following dependent sum:

$$\mathsf{M}^*.\mathsf{type} := (\sigma : \mathbf{\Phi} \Longrightarrow \mathsf{M}.\mathsf{type}) \times \{\mathcal{U} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M}.\mathsf{el}\,\sigma\} \tag{1}$$

Unfortunately, the definition above does not go through because it is not the case that the dependent sum above restricts in object-space to $\mathsf{M}.\mathsf{type}$! But it is instructive to check how close we got; in particular, within object-space (*i.e.* assuming $\mathbf{\Phi} = \top$) the projection function $\pi_1 : \mathsf{M}^*.\mathsf{type} \longrightarrow \mathsf{M}.\mathsf{type}$ restricts in object-space to an isomorphism, *i.e.* the following proposition holds:

$$\mathbf{\Phi} \Longrightarrow \forall \sigma : \mathsf{M}.\mathsf{type}. \exists! \sigma^* : \mathsf{M}^*.\mathsf{type}. \pi_1\, \sigma^* = \sigma$$

Proof. Assuming $\mathbf{\Phi} = \top$ and an element $\sigma : \mathsf{M}.\mathsf{type}$, it suffices to check that the collection of elements $\sigma^* : \mathsf{M}^*.\mathsf{type}$ such that $\pi_1\, \sigma^* = \sigma$ is a singleton; equivalently, that there exists only one $\sigma' : \{\mathcal{U} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M}.\mathsf{el}\, \sigma\}$. Because we are in object-space, this we have $\sigma' : \{\mathcal{U} \mid \top \hookrightarrow \mathsf{M}.\mathsf{el}\, \sigma\}$ which is by definition the collection of elements of $\mathcal{U}$ that are equal to $\mathsf{M}.\mathsf{el}\, \sigma$, *i.e.* the singleton $\{\mathsf{M}.\mathsf{el}\, \sigma\}$. □

Remark 16. The essence of the proof above is that each type $\{A \mid \mathbf{\Phi} \hookrightarrow a\}$ is $\mathbf{\Phi}$-sealed, so it restricts within object-space to a singleton. We will use this insight to abstract the problem above into a solution to all similar problems via a generalization of *refinement types*.

Thus what is missing is a way to construct a dependent sum like $\mathsf{M}^*.\mathsf{type}$ in such a way that the object-space isomorphism verified above is actually the *identity function*. This is the purpose of the generalized ***refinement type*** connective that we introduce below. Put simply, the refinement types are special versions of dependent sums where the index comes from object-space and the fibers lie in meta-space; what distinguishes a refinement type from a dependent sum is that within object-space, it becomes *equal* to its index type.

We introduce refinement types by describing their rules of inference informally. The formation rule takes an object-space type $A : \mathbf{\Phi} \Longrightarrow \mathcal{U}$ and a meta-space family of types $B : (\mathbf{\Phi} \Longrightarrow A) \longrightarrow \mathcal{U}$ (*i.e.* such that each Bx is $\mathbf{\Phi}$-sealed) to a type $[\mathbf{\Phi} \hookrightarrow x : A \mid Bx]$ that restricts within object space to A:

$$\frac{A : \mathbf{\Phi} \Longrightarrow \mathcal{U} \quad B : (\mathbf{\Phi} \Longrightarrow A) \longrightarrow \mathcal{U} \quad x : \mathbf{\Phi} \Longrightarrow A \vdash Bx \text{ is } \mathbf{\Phi}\text{-sealed}}{[\mathbf{\Phi} \hookrightarrow x : A \mid Bx] : \mathcal{U}}$$

$$\frac{A : \mathbf{\Phi} \Longrightarrow \mathcal{U} \quad B : (\mathbf{\Phi} \Longrightarrow A) \longrightarrow \mathcal{U} \quad x : \mathbf{\Phi} \Longrightarrow A \vdash Bx \text{ is } \mathbf{\Phi}\text{-sealed} \quad \mathbf{\Phi} = \top}{[\mathbf{\Phi} \hookrightarrow x : A \mid Bx] = A : \mathcal{U}}$$

What we mean by "restricts within object space to A" is that when $\mathbf{\Phi} = \top$, we have a definitional equality $[\mathbf{\Phi} \hookrightarrow x : A \mid Bx] = A$. Our intention is that aside from this special equality rule, the refinement type $[\mathbf{\Phi} \hookrightarrow x : A \mid Bx]$ should behave like the dependent sum $(x : \mathbf{\Phi} \Longrightarrow A) \times Bx$.

We next give the introduction rule, which will mirror the introduction rule for the dependent sum; we must also provide a rule that for what the introduction form becomes when restricted into object-space:

$$\frac{a : \mathbf{\Phi} \Longrightarrow A \quad b : Ba}{[\mathbf{\Phi} \hookrightarrow a \mid b] : [\mathbf{\Phi} \hookrightarrow x : A \mid Bx]} \qquad \frac{a : \mathbf{\Phi} \Longrightarrow A \quad b : Ba \quad \mathbf{\Phi} = \top}{[\mathbf{\Phi} \hookrightarrow a \mid b] = a : A}$$

Next we describe the elimination forms; for a dependent sum, we ordinarily have both first and second projections, but in our case the first projection is automatically given by virtue of the equations we have imposed. To see that this is the case, we show that we can already construct the first projection $\mathsf{fst} : [\mathbf{\Phi} \hookrightarrow x : A \mid Bx] \longrightarrow \mathbf{\Phi} \Longrightarrow A$. Given $u : [\mathbf{\Phi} \hookrightarrow x : A \mid Bx]$, we must construct an element $\mathsf{fst}\, u : \mathbf{\Phi} \Longrightarrow A$, *i.e.* an element of A assuming $\mathbf{\Phi} = \top$; under this assumption, we have $[\mathbf{\Phi} \hookrightarrow x : A \mid Bx] = A$ and hence we may choose $\mathsf{fst}\, u := u$. On the other hand, we must provide an explicit elimination form for the second projection, which we shall write using an underline:

$$\frac{u : [\mathbf{\Phi} \hookrightarrow x : A \mid Bx]}{\underline{u} : Bu}$$

Definition 17. *Given a refined element $u : [\mathbf{\Phi} \hookrightarrow x : A \mid Bx]$, we will refer to $\underline{u} : Bu$ as the* ***refinement datum*** *of u.*

The β- and η-laws of the refinement type are given below:

$$\frac{a : \Phi \Longrightarrow A \quad b : Ba}{\underline{[\Phi \hookrightarrow a \mid b]} = b : Ba} \qquad \frac{u : [\Phi \hookrightarrow x : A \mid Bx]}{u = [\Phi \hookrightarrow u \mid \underline{u}] : [\Phi \hookrightarrow x : A \mid Bx]}$$

Example 18. Our first application of the refinement type is to give a well-typed definition of $\mathsf{M}^*.\mathsf{type}$:

$$\mathsf{M}^*.\mathsf{type} := [\Phi \hookrightarrow \sigma : \mathsf{M}.\mathsf{type} \mid \{\mathcal{U} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,\sigma\}]$$

1.5.3 Constructing the synthetic logical predicate

We are now equipped to define the synthetic logical predicate.

begin structure $\mathsf{M}^* : \{\mathsf{STLC}_{\mathcal{V}} \mid \Phi \hookrightarrow \mathsf{M}\}$
  $\mathsf{type} := [\Phi \hookrightarrow \sigma : \mathsf{M}.\mathsf{type} \mid \{\mathcal{U} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,\sigma\}]$
  $\mathsf{el}\,\sigma := \underline{\sigma}$

Computation 19. (Decoding) It is instructive to understand why our definition of el above is correct. Explicitly, we need an element $\mathsf{el} : \{\mathsf{type} \longrightarrow \mathcal{V} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\}$; we compute a solution "interactively" in several steps:

1. To fill the hole $?_0 : \{\mathsf{type} \longrightarrow \mathcal{V} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\}$, it suffices to fix an element $\sigma : \mathsf{type}$ and pose $?_1\sigma : \{\mathcal{V} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,\sigma\}$, solving $?_0 := \lambda\sigma.?_1\sigma$.
2. By cumulativity of $\mathcal{U} \subseteq \mathcal{V}$, it suffices to pose $?_2\sigma : \{\mathcal{U} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,\sigma\}$ and solve $?_1\sigma := ?_2\sigma$.
3. Projecting out the refinement datum, it suffices to pose $?_3\sigma : \{[\Phi \hookrightarrow \sigma : \mathsf{M}.\mathsf{type} \mid \{\mathcal{U} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,\sigma\}] \mid \Phi \hookrightarrow \sigma\}$ and solve $?_2\sigma := \underline{?_3\sigma}$.
4. We may solve $?_3\sigma := \sigma$, as the type of $?_3\sigma$ is in fact equal to $\{\mathsf{type} \mid \Phi \hookrightarrow \sigma\}$ and any element (by definition) restricts to itself on object-space.
5. Thus by unravelling the above, we have solved $?_0 := \lambda\sigma.\underline{\sigma}$.

Computation 20. (Function types) We will interpret the function type connective step-by-step as well. We need an element $\mathsf{arr} : \{\mathsf{type} \longrightarrow \mathsf{type} \longrightarrow \mathsf{type} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{arr}\}$. We step through the construction process a bit more rapidly until we arrive at a goal that requires some thought:

$$\begin{aligned}
&\mathsf{arr} := ? : \{\mathsf{type} \longrightarrow \mathsf{type} \longrightarrow \mathsf{type} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{arr}\}\\
&\mathsf{arr}\ \sigma\ \tau := ? : \{\mathsf{type} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{arr}\ \sigma\ \tau\}\\
&\mathsf{arr}\ \sigma\ \tau := ? : \{[\Phi \hookrightarrow \sigma : \mathsf{M}.\mathsf{type} \mid \{\mathcal{U} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,\sigma\}] \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{arr}\ \sigma\ \tau\}\\
&\mathsf{arr}\ \sigma\ \tau := [\Phi \hookrightarrow (?_0 : \{\mathsf{M}.\mathsf{type} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{arr}\ \sigma\ \tau\}) \mid (?_1 : \{\mathcal{U} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,?_0\})]\\
&\mathsf{arr}\ \sigma\ \tau := [\Phi \hookrightarrow \mathsf{M}.\mathsf{arr}\ \sigma\ \tau \mid (?_1 : \{\mathcal{U} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{el}\,(\mathsf{M}.\mathsf{arr}\ \sigma\ \tau)\})]
\end{aligned}$$

How do we choose the correct element of $\mathcal{U}$ to fill the hole $?_1$ above? We know that ultimately have to exhibit λ-abstraction, application, β- and η-laws; but up to isomorphism, there can be only one connective that supports all this structure: the function space itself. Thus we know in advance that we are required to choose an element of $\mathcal{U}$ representing the function space $\underline{\sigma} \longrightarrow \underline{\tau}$ that restricts in object-space to the function space of the generic structure M. This is a job for the refinement type!

$$?_1 := [\Phi \hookrightarrow e : \mathsf{M}.\mathsf{el}\,(\mathsf{M}.\mathsf{arr}\ \sigma\ \tau) \mid \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{app}\ e\ -\}]$$

The use of the refinement type above is valid because each type $\{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{app}\ e\ x\ -\}$ is Φ-sealed. Thus in full, we complete our definition of the function space using an auxiliary definition:

$$\begin{aligned}
&\mathsf{arr}'\ \sigma\ \tau := [\Phi \hookrightarrow e : \mathsf{M}.\mathsf{el}\,(\mathsf{M}.\mathsf{arr}\ \sigma\ \tau) \mid \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow \mathsf{M}.\mathsf{app}\ e\ -\}]\\
&\mathsf{arr}\ \sigma\ \tau := [\Phi \hookrightarrow \mathsf{M}.\mathsf{arr}\ \sigma\ \tau \mid \mathsf{arr}'\ \sigma\ \tau]
\end{aligned}$$

We define the λ-abstraction as follows; make sure you understand each step below:

$$\begin{aligned}
&\mathsf{lam}\ u := ? : \{\underline{\mathsf{arr}\ \sigma\ \tau} \mid \Phi \hookrightarrow \mathsf{M.lam}\ u\} \\
&\mathsf{lam}\ u := ? : \{\mathsf{arr}'\ \sigma\ \tau \mid \Phi \hookrightarrow \mathsf{M.lam}\ u\} \\
&\mathsf{lam}\ u := ? : \{[\Phi \hookrightarrow e : \mathsf{M.el}\,(\mathsf{M.arr}\ \sigma\ \tau) \mid \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow \mathsf{M.app}\ e\ -\}] \mid \Phi \hookrightarrow \mathsf{M.lam}\ u\} \\
&\mathsf{lam}\ u := [\Phi \hookrightarrow (?_0 : \{\mathsf{M.el}\,(\mathsf{M.arr}\ \sigma\ \tau) \mid \Phi \hookrightarrow \mathsf{M.lam}\ u\}) \mid (?_1 : \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow \mathsf{M.app}\ ?_0\ -\})] \\
&\mathsf{lam}\ u := [\Phi \hookrightarrow \mathsf{M.lam}\ u \mid (?_1 : \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow \mathsf{M.app}\,(\mathsf{M.lam}\ u)\ -\})] \\
&\mathsf{lam}\ u := [\Phi \hookrightarrow \mathsf{M.lam}\ u \mid (?_1 : \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow u\})] \\
&\mathsf{lam}\ u := [\Phi \hookrightarrow \mathsf{M.lam}\ u \mid \lambda x.(?_1 : \{\underline{\tau} \mid \Phi \hookrightarrow u\,x\})] \\
&\mathsf{lam}\ u := [\Phi \hookrightarrow \mathsf{M.lam}\ u \mid u]
\end{aligned}$$

The application operator is likewise easy to define by projecting the refinement datum of an element $u : \underline{\mathsf{arr}\ \sigma\ \tau}$, recalling that $\underline{\mathsf{arr}\ \sigma\ \tau} = [\Phi \hookrightarrow e : \mathsf{M.el}\,(\mathsf{M.arr}\ \sigma\ \tau) \mid \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \Phi \hookrightarrow \mathsf{M.app}\ e\ -\}]$.

$$\begin{aligned}
&\mathsf{app}\ u\ v := ? : \{\underline{\tau} \mid \Phi \hookrightarrow \mathsf{M.app}\ u\ v\} \\
&\mathsf{app}\ u\ v := \underline{u}\ v
\end{aligned}$$

The computation rules can be seen to hold by virtue of $\mathsf{M.arr}_\beta$, $\mathsf{M.arr}_\eta$, and the β/η laws of the refinement type connective:

$$\mathsf{app}\,(\mathsf{lam}\ u)\ v = \underline{\mathsf{lam}\ u}\ v = \underline{[\Phi \hookrightarrow \mathsf{M.lam}\ u \mid u]}\ v = \underline{[\Phi \hookrightarrow \mathsf{M.lam}\ u \mid u]}\ v = u\,v$$

$$\mathsf{lam}\,(\lambda x.\mathsf{app}\ u\ x) = [\Phi \hookrightarrow \mathsf{M.lam}\,(\lambda x.\mathsf{M.app}\ u\ x) \mid \lambda x.\underline{u}\ x] = [\Phi \hookrightarrow u \mid \underline{u}] = u$$

Thus we conclude:

$$\begin{aligned}
&\mathsf{arr}_\beta := \star \\
&\mathsf{arr}_\eta u := \star
\end{aligned}$$

Computation 21. (Product types) The product type is interpreted using essentially the same recipe as the function type: we refine the object-space function type $\mathsf{M.prod}\ \sigma\ \tau$ by the ambient product type $\underline{\sigma} \times \underline{\tau}$.

$$\begin{aligned}
&\mathsf{prod}'\ \sigma\ \tau := [\Phi \hookrightarrow e : \mathsf{M.el\,M.prod}\ \sigma\ \tau \mid \{\underline{\sigma} \times \underline{\tau} \mid \Phi \hookrightarrow (\mathsf{M.fst}\ e, \mathsf{M.snd}\ e)\}] \\
&\mathsf{prod}\ \sigma\ \tau := [\Phi \hookrightarrow \mathsf{M.prod}\ \sigma\ \tau \mid \mathsf{prod}'\ \sigma\ \tau] \\
&\mathsf{pair}\ u\ v := [\Phi \hookrightarrow \mathsf{M.pair}\ u\ v \mid (u, v)] \\
&\mathsf{fst}\ u := \pi_1\ \underline{u} \\
&\mathsf{snd}\ u := \pi_2\ \underline{u} \\
&\mathsf{prod}_{\beta 1} := \star \\
&\mathsf{prod}_{\beta 2} := \star \\
&\mathsf{prod}_\eta u := \star
\end{aligned}$$

Computation 22. (Booleans) The most important part of the synthetic logical predicate is to interpret the type of booleans. Our goal is to show that for any object-boolean, its closed instances are exactly M.true and M.false, so this must be reflected in our definition of bool. We have already stipulated that the closed elements of a type are accessed synthetically using by projecting its meta-space component $\Phi \bullet (\Phi \Longrightarrow \mathsf{M.el\,M.bool})$.

Let $\mathbf{2} := \mathbf{1} + \mathbf{1}$ be the type with two elements $0 := \mathsf{inl}\ \star$ and $1 := \mathsf{inr}\ \star$. We may define an enumeration of the (desired) canonical booleans as follows:

$$\begin{aligned}
&\mathsf{enum} : \mathbf{2} \longrightarrow \Phi \Longrightarrow \mathsf{M.el\,M.bool} \\
&\mathsf{enum}\ 0 := \mathsf{M.true} \\
&\mathsf{enum}\ 1 := \mathsf{M.false}
\end{aligned}$$

We then define the logical predicate on the booleans to associate with every object-boolean a element of the corresponding fiber of enum:

$$\begin{aligned}
&\mathsf{bool} := ? : \{\mathsf{type} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M.bool}\}\\
&\mathsf{bool} := [\mathbf{\Phi} \hookrightarrow \mathsf{M.bool} \mid (? : \{\mathcal{U} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M.el\ M.bool}\})]\\
&\mathsf{bool} := [\mathbf{\Phi} \hookrightarrow \mathsf{M.bool} \mid [\mathbf{\Phi} \hookrightarrow x : \mathsf{M.el\ M.bool} \mid \mathbf{\Phi} \bullet \{i : \mathbf{2} \mid \mathsf{enum}\ i = x\}]]
\end{aligned}$$

Reading the definition above, an element of $\underline{\mathsf{bool}}$ is thus an object boolean together with a proof that its ***generic closed instance*** is either $\eta_{\mathbf{\Phi}}.\mathsf{true}$ or $\eta_{\mathbf{\Phi}}.\mathsf{false}$. It is not difficult to interpret the boolean constants in the logical predicate now:

$$\begin{aligned}
&\mathsf{true} := [\mathbf{\Phi} \hookrightarrow \mathsf{M.true} \mid \eta_{\mathbf{\Phi}}.0]\\
&\mathsf{false} := [\mathbf{\Phi} \hookrightarrow \mathsf{M.false} \mid \eta_{\mathbf{\Phi}}.1]
\end{aligned}$$

The case expression is defined by pattern matching:

$$\begin{aligned}
&\mathsf{case}\ [\mathbf{\Phi} \hookrightarrow \mathsf{M.true} \mid \eta_{\mathbf{\Phi}}.0]\ v\ w := v\\
&\mathsf{case}\ [\mathbf{\Phi} \hookrightarrow \mathsf{M.false} \mid \eta_{\mathbf{\Phi}}.1]\ v\ w := w\\
&\mathsf{case}\ [\mathbf{\Phi} \hookrightarrow u \mid \star]\ v\ w := \mathsf{M.case}\ u\ v\ w
\end{aligned}$$

The final case is well-defined because we have $\mathbf{\Phi} = \top$ in scope; the β-rules hold by definition:

$$\begin{aligned}
&\mathsf{bool}_{\beta 1} := \star\\
&\mathsf{bool}_{\beta 2} := \star
\end{aligned}$$

Finally we check that the η-law holds, fixing $u : \underline{\mathsf{bool}} \longrightarrow \underline{\sigma}$. We need to check that $u = \lambda x.\, \mathsf{case}\ x\ (u\,\mathsf{true})\ (u\,\mathsf{false})$; we may fix $x : \underline{\mathsf{bool}}$ to check that $ux = \mathsf{case}\ x\ (u\,\mathsf{true})\ (u\,\mathsf{false}) : \underline{\sigma}$. First of all, this equation is a $\mathbf{\Phi}$-sealed proposition because it restricts in object-space to our assumption $\mathsf{M.bool}_{\eta}$; by the universal property of the meta-space component (Lemma 10), we may assume that there exists some $i : \mathbf{2}$ such that $\mathbf{\Phi} \Longrightarrow \mathsf{enum}\ i = x$ and $\underline{x} = \eta_{\mathbf{\Phi}}.i$ and thus $x = [\mathbf{\Phi} \hookrightarrow \mathsf{enum}\ i \mid \eta_{\mathbf{\Phi}}.i]$. We proceed by cases on i:

1. If $i = 0$, then $x = \mathsf{true}$ and thus $ux = \mathsf{case}\ x\ (u\,\mathsf{true})\ (u\,\mathsf{false})$.
2. If $i = 1$, then $x = \mathsf{false}$ and thus $ux = \mathsf{case}\ x\ (u\,\mathsf{true})\ (u\,\mathsf{false})$ likewise.

Thus we conclude:

$$\begin{aligned}
&\quad \mathsf{bool}_{\eta}\ u := \star\\
&\textbf{end structure}\ \mathsf{M}^*
\end{aligned}$$

1.5.4 The full synthetic logical predicate

We summarize the entire logical predicate below:

$$\begin{aligned}
&\textbf{begin structure}\ \mathsf{M}^* : \{\mathsf{STLC}_{\mathcal{V}} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M}\}\\
&\quad \mathsf{type} := [\mathbf{\Phi} \hookrightarrow \sigma : \mathsf{M.type} \mid \{\mathcal{U} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M.el}\ \sigma\}]\\
&\quad \mathsf{el}\ \sigma := \underline{\sigma}\\
&\\
&\quad \mathsf{arr}\ \sigma\ \tau := [\mathbf{\Phi} \hookrightarrow \mathsf{M.arr}\ \sigma\ \tau \mid [\mathbf{\Phi} \hookrightarrow e : \mathsf{M.el}\ (\mathsf{M.arr}\ \sigma\ \tau) \mid \{\underline{\sigma} \longrightarrow \underline{\tau} \mid \mathbf{\Phi} \hookrightarrow \mathsf{M.app}\ e\ x\}]]\\
&\quad \mathsf{lam}\ u := [\mathbf{\Phi} \hookrightarrow \mathsf{M.lam}\ u \mid u]\\
&\quad \mathsf{app}\ u\ v := \underline{u}\ v\\
&\quad \mathsf{arr}_{\beta} := \star\\
&\quad \mathsf{arr}_{\eta} u := \star\\
&\\
&\quad \mathsf{prod}\ \sigma\ \tau := [\mathbf{\Phi} \hookrightarrow \mathsf{M.prod}\ \sigma\ \tau \mid [\mathbf{\Phi} \hookrightarrow e : \mathsf{M.el\ M.prod}\ \sigma\ \tau \mid \{\underline{\sigma} \times \underline{\tau} \mid \mathbf{\Phi} \hookrightarrow (\mathsf{M.fst}\ e, \mathsf{M.snd}\ e)\}]]\\
&\quad \mathsf{pair}\ u\ v := [\mathbf{\Phi} \hookrightarrow \mathsf{M.pair}\ u\ v \mid (u, v)]\\
&\quad \mathsf{fst}\ u := \pi_1\ \underline{u}
\end{aligned}$$

$\mathsf{snd}\ u := \pi_2\ \underline{u}$
$\mathsf{prod}_{\beta 1} := \star$
$\mathsf{prod}_{\beta 2} := \star$
$\mathsf{prod}_{\eta}\ u := \star$

$\mathsf{bool} := [\mathbf{\Phi} \hookrightarrow \mathsf{M.bool} \mid [\mathbf{\Phi} \hookrightarrow x : \mathsf{M.el\ M.bool} \mid \mathbf{\Phi} \bullet \{ i : \mathbf{2} \mid \mathsf{enum}\ i = x\}]]$
$\mathsf{true} := [\mathbf{\Phi} \hookrightarrow \mathsf{M.true} \mid \eta_{\mathbf{\Phi}}.0]$
$\mathsf{false} := [\mathbf{\Phi} \hookrightarrow \mathsf{M.false} \mid \eta_{\mathbf{\Phi}}.1]$
$\mathsf{case}\ [\mathbf{\Phi} \hookrightarrow \mathsf{M.true} \mid \eta_{\mathbf{\Phi}}.0]\ v\ w := v$
$\mathsf{case}\ [\mathbf{\Phi} \hookrightarrow \mathsf{M.false} \mid \eta_{\mathbf{\Phi}}.1]\ v\ w := w$
$\mathsf{case}\ [\mathbf{\Phi} \hookrightarrow u \mid \star]\ v\ w := \mathsf{M.case}\ u\ v\ w$
$\mathsf{bool}_{\beta 1} := \star$
$\mathsf{bool}_{\beta 2} := \star$
$\mathsf{bool}_{\eta}\ u := \star$

end structure M^*

1.6 Canonicity for typed λ-calculus

We set out to prove a theorem about the actual syntax of the actual typed λ-calculus; but so far we have seen nothing "actual" whatsoever. In this section, we specify the actual syntax of typed λ-calculus as a meta-space inductive definition, and prove the canonicity theorem relative to a ***synthetic adequacy*** postulate that asserts a completeness property for the interpretation of "actual" syntax into the generic model M.

1.6.1 The syntax of typed λ-calculus, analytically

The typed λ-calculus can be presented by ordinary syntax with several forms of judgment:

1. The judgment $\Gamma\ \mathit{ctx}$ means that Γ is a context.
2. The judgment $\sigma\ \mathit{type}$ means that A is a type.
3. The judgment $\Gamma \vdash e : \sigma$ means that M is a term of type σ in context Γ.
4. The judgment $\Gamma \vdash e \equiv e' : \sigma$ means that e and e' are definitionally equal terms of type σ in context Γ.

In the synthetic world, meta-space corresponds to ordinary (external) set theory; thus it is natural to formalize the analytic syntax as an inductive definition within meta-space. Meta space inductive definitions are just like ordinary inductive definitions, except that their induction principles apply only to meta-space motives, *i.e.* motives that are $\mathbf{\Phi}$-sealed.

To get started, we give an inductive definition of the meta-space type of analytic types $\mathtt{Type}$.

$$\frac{}{\mathtt{bool} : \mathtt{Type}} \qquad \frac{\sigma, \tau : \mathtt{Type}}{\sigma \Longrightarrow \tau : \mathtt{Type}} \qquad \frac{\sigma, \tau : \mathtt{Type}}{\sigma \times \tau : \mathtt{Type}}$$

Remark 23. The universal property of $\mathtt{Type}$ is exactly that of an inductive type, except the motive $C : \mathtt{Type} \longrightarrow \mathcal{U}$ must lie in meta-space, in the sense that each $C\sigma$ is $\mathbf{\Phi}$-sealed. Advanced readers may be curious how to explictly construct such a meta-space inductive type; this can be done by means of an ordinary *quotient inductive type*, whose generators are the ones above extended as follows:

$$\frac{\mathbf{\Phi} = \top}{\star : \mathtt{Type}} \qquad \frac{\mathbf{\Phi} = \top \quad \sigma : \mathtt{Type}}{\sigma = \star : \mathtt{Type}}$$

We can define the meta-space type $\mathtt{Ctx}$ of analytic contexts as follows, fixing some infinite decidable meta-space type of names $\mathtt{Name}$ (*e.g.*, integers or the set of unicode symbols, *etc.*):

$$\frac{}{\cdot : \mathtt{Ctx}} \qquad \frac{\Gamma : \mathtt{Ctx} \quad x : \mathtt{Name} \quad \sigma : \mathtt{Type}}{(\Gamma, x : \sigma) : \mathtt{Ctx}}$$

We can define the meta-space set of variables bound by a context $\mathtt{Vars}(\Gamma) \subseteq \mathtt{Name}$:

$$\begin{aligned} \mathtt{Vars}(\cdot) &:= \emptyset \\ \mathtt{Vars}(\Gamma, x : \sigma) &:= \mathtt{Vars}(\Gamma) \cup \{x\} \end{aligned}$$

Finally we define the meta-space indexed inductive type of analytic *preterms* $\mathtt{Preterm}$; we do not belabor this definition, but instead give only a couple representative examples:

$$\frac{}{\mathtt{true} : \mathtt{Preterm}} \qquad \frac{x : \mathtt{Name} \quad \sigma : \mathtt{Type} \quad M : \mathtt{Preterm}}{\lambda x^{\sigma}.e : \mathtt{Preterm}} \qquad \frac{f, e : \mathtt{Preterm}}{f(e) : \mathtt{Preterm}}$$

It is easy to define renamings of bound variables, which we may use to define the relation of α-equivalence $\equiv_{\alpha}$ in the standard way à la Barendregt; we will write $\mathtt{Preterm}/\equiv_{\alpha}$ for the collection of α-equivalence classes of preterms. Informally we will not distinguish between a preterm and its α-equivalence class. We can also define the set of free variables $\mathtt{FV}(e)$ of a given equivalence class of preterms, again following Barendregt; we may also define substitution of terms for free variables $[e/x]f$ by recursion.

Next we may define the meta-space indexed inductive definition of well-typed terms $(\Gamma \vdash M : \sigma)$ where $\Gamma : \mathtt{Ctx}$ and $M, N : \mathtt{Preterm}/\equiv_{\alpha}$ and $\sigma : \mathtt{Type}$, subject to the constraint that $\mathtt{FV}(M) \subseteq \mathtt{Vars}(\Gamma)$. For instance, we will include generators like the following:

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau \quad (x \notin \mathtt{Vars}(\Gamma))}{\lambda x^{\sigma}.e : \sigma \Rightarrow \tau} \qquad \frac{\Gamma \vdash f : \sigma \Rightarrow \tau \quad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

It is possible to prove by induction that substitution preserves well-typedness. Finally, we define the meta-space indexed inductive definition of definitional equivalence $(\Gamma \vdash e \equiv e' : \sigma)$ where $\Gamma : \mathtt{Ctx}$ and $\sigma : \mathtt{Type}$ and $\Gamma \vdash e, e' : \sigma$. In addition to the β- and η-laws, we must also include generators to ensure that definitional equivalence is a symmetric congruence. For instance:

$$\frac{}{\Gamma \vdash \mathtt{true} \equiv \mathtt{true} : \mathtt{bool}} \qquad \frac{\Gamma, x : \sigma \vdash e \equiv e' : \tau}{\Gamma \vdash \lambda x^{\sigma}.e \equiv \lambda x^{\sigma}.e' : \sigma \Rightarrow \tau} \qquad \frac{\Gamma \vdash f \equiv f' : \sigma \Rightarrow \tau \quad \Gamma \vdash e \equiv e' : \sigma}{\Gamma \vdash f(e) \equiv f'(e') : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash f : \tau \quad \Gamma \vdash e : \sigma}{\Gamma \vdash (\lambda x^{\sigma}.f)(e) \equiv [e/x]f : \tau} \qquad \frac{\Gamma, x : \sigma \vdash f : \tau \quad \Gamma \vdash e : \sigma}{\Gamma \vdash [e/x]f \equiv (\lambda x^{\sigma}.f)(e) : \tau}$$

We may prove by induction that definitional equivalence is in fact transitive, thus it is an equivalence relation. This concludes the definition of the *analytic* syntax of typed λ-calculus in meta-space.

1.6.2 The interpretation function corresponding to a synthetic model

With the analytic syntax of typed λ-calculus in hand, our next step is to define a meta-space *interpretation function* into any synthetic model of typed λ-calculus, *i.e.* any structure $\mathsf{N} : \mathsf{STLC}_{\mathcal{U}}$. We must exercise care in specifying this interpretation, because the structure N need not lie within meta-space, and yet the analytic syntax that we have defined is sealed within meta-space.

For each analytic type $\sigma : \mathtt{Type}$, we will choose an element $[\![\sigma]\!]^{\mathsf{N}} : \mathbf{\Phi} \bullet \mathcal{U}$ as follows:

$$\begin{aligned} [\![\mathtt{bool}]\!]^{\mathsf{N}} &:= \eta_{\mathbf{\Phi}}.\mathsf{N.bool} \\ [\![\sigma \Rightarrow \tau]\!]^{\mathsf{N}} &:= \{\sigma \leftarrow [\![\sigma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; \eta_{\mathbf{\Phi}}.(\mathsf{N.arr}\ \sigma\ \tau)\} \end{aligned}$$

The above is enough for us to define the type of environments $[\![\Gamma]\!]^{\mathsf{N}} : \mathbf{\Phi} \bullet \mathcal{U}$ within metaspace for any analytic context $\Gamma : \mathtt{Ctx}$.

$$\begin{aligned} [\![\cdot]\!]^{\mathsf{N}} &:= \eta_{\mathbf{\Phi}}.\mathbf{1} \\ [\![\Gamma, x : \sigma]\!]^{\mathsf{N}} &:= \{\Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \sigma \leftarrow [\![\sigma]\!]^{\mathsf{N}}; \eta_{\mathbf{\Phi}}.(\Gamma \times \mathsf{N.el}\ \sigma)\} \end{aligned}$$

Finally for each analytic term $\Gamma \vdash e : \tau$, we will choose an element with the following metaspace type:

$$[\![\Gamma \vdash e : \tau]\!]^{\mathsf{N}} : \mathbf{\Phi} \bullet [\Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}, \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}].\, \Gamma \to \mathsf{N.el}\ \tau$$

We will give a couple cases of this interpretation function. For instance, we derive the interpretation of the true boolean as follows:

$$\begin{aligned} [\![\Gamma \vdash \mathtt{true} : \mathtt{bool}]\!]^{\mathsf{N}} &:= ? : \mathbf{\Phi} \bullet [\Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}, \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}].\, \Gamma \to \mathsf{N.el}\ \tau \\ &:= \{\Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; (? : \mathbf{\Phi} \bullet (\Gamma \to \mathsf{N.el}\ \tau))\} \\ &:= \{\Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; \eta_{\mathbf{\Phi}}.(\lambda \gamma.\mathsf{N.true})\} \end{aligned}$$

The case for λ-abstraction is a bit more complex, as it involves an induction hypothesis.

$$\begin{aligned} [\![\Gamma \vdash \lambda x^{\sigma}.e : \sigma \Rightarrow \tau]\!]^{\mathsf{N}} &:= ? : \mathbf{\Phi} \bullet [\Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \rho \leftarrow [\![\sigma \Rightarrow \tau]\!]^{\mathsf{N}}].\, \Gamma \to \mathsf{N.el}\ \rho \\ &:= \begin{cases} \Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \sigma \leftarrow [\![\sigma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; \\ ? : \mathbf{\Phi} \bullet (\Gamma \to \mathsf{N.el}\ (\mathsf{N.arr}\ \sigma\ \tau)) \end{cases} \\ &:= \begin{cases} \Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \sigma \leftarrow [\![\sigma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; \\ ? : \mathbf{\Phi} \bullet (\Gamma \to \mathsf{N.el}\ (\mathsf{N.arr}\ \sigma\ \tau)) \end{cases} \\ &:= \begin{cases} \Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \sigma \leftarrow [\![\sigma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; \\ e \leftarrow [\![\Gamma, x : \sigma \vdash e : \tau]\!]^{\mathsf{N}}; \\ ? : \mathbf{\Phi} \bullet (\Gamma \to \mathsf{N.el}\ (\mathsf{N.arr}\ \sigma\ \tau)) \end{cases} \\ &:= \begin{cases} \Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \sigma \leftarrow [\![\sigma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; \\ e \leftarrow [\![\Gamma, x : \sigma \vdash e : \tau]\!]^{\mathsf{N}}; \\ \eta_{\mathbf{\Phi}}.(\lambda \gamma.\mathsf{N.lam}\ (\lambda x.e(\gamma, x))) \end{cases} \end{aligned}$$

We conclude our sketch by deriving the intrepretation of application expressions, given $\Gamma \vdash f : \sigma \Rightarrow \tau$ and $\Gamma \vdash e : \sigma$.

$$\begin{aligned} [\![\Gamma \vdash f(e) : \tau]\!]^{\mathsf{N}} &:= ? : \mathbf{\Phi} \bullet [\Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}].\, \Gamma \to \mathsf{N.el}\ \tau \\ &:= \begin{cases} \Gamma \leftarrow [\![\Gamma]\!]^{\mathsf{N}}; \tau \leftarrow [\![\tau]\!]^{\mathsf{N}}; \sigma \leftarrow [\![\sigma]\!]^{\mathsf{N}}; \\ f \leftarrow [\![\Gamma \vdash f : \sigma \Rightarrow \tau]\!]^{\mathsf{N}}; \\ e \leftarrow [\![\Gamma \vdash e : \sigma]\!]^{\mathsf{N}}; \\ \eta_{\mathbf{\Phi}}.(\lambda \gamma.\mathsf{M.app}\ (f\gamma)\ (e\gamma)) \end{cases} \end{aligned}$$

Lemma 24. (Soundness) *If* $\Gamma \vdash e \equiv e' : \tau$ *then* $[\![\Gamma \vdash e : \tau]\!]^{\mathsf{N}} = [\![\Gamma \vdash e' : \tau]\!]^{\mathsf{N}}$.

Proof. By structural induction. □

1.6.3 The synthetic adequacy postulate and the proof of canonicity

We finally introduce the last postulate needed to state and prove the canonicity theorem in the synthetic setting. This axiom, called the ***synthetic adequacy*** postulate, expresses the sense in which the synthetic λ-terms as embodied by the generic model M are related to the "actual" analytic syntax of typed λ-calculus.

Let $\mathsf{M} : \mathbf{\Phi} \Longrightarrow \mathsf{STLC}_{\mathcal{U}}$ be the generic object-space λ-structure; we will write $(\mathbf{\Phi} \Longrightarrow \mathsf{M}) : \mathsf{STLC}_{\mathcal{U}}$ for the structure obtained by extending its sorts from object-space, *i.e.* $(\mathbf{\Phi} \Longrightarrow \mathsf{M}).\mathsf{type} = (\mathbf{\Phi} \Longrightarrow \mathsf{M}.\mathsf{type})$. Then the synthetic adequacy postulate is a converse to Lemma 24 for $\mathsf{N} = (\mathbf{\Phi} \Longrightarrow \mathsf{M})$.

Postulate. (Synthetic adequacy) For any analytic terms $\Gamma \vdash e, e' : \tau$ we have $\Gamma \vdash e \equiv e' : \tau$ if $[\![\Gamma \vdash e : \tau]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}} = [\![\Gamma \vdash e' : \tau]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}}$.

We can now state and prove the synthetic canonicity theorem, which is stated in the logic of meta-space.

Theorem 25. (Synthetic canonicity) *Within meta-space, if* $\cdot \vdash e : \mathtt{bool}$ *then either* $\cdot \vdash e \equiv \mathtt{true} : \mathtt{bool}$ *or* $\cdot \vdash e \equiv \mathtt{false} : \mathtt{bool}$ *but not both.*

Before we prove Theorem 25, we must explain what we mean by the qualifier "within meta-space". The problem lies with the interpretation of "but not both": if P is a meta-space proposition, it will not be the case that the naïve negation $\neg P = P \longrightarrow \bot$ is a meta-space proposition. As meta-space is meant to be the complement of object-space, it turns out that the correct way to negate a meta-space proposition is $P \longrightarrow \mathbf{\Phi}$. Thus the formal statement of Theorem 25 is the conjunction of the following two statements:

$$(\cdot \vdash \mathtt{true} \equiv \mathtt{false} : \mathtt{bool}) \longrightarrow \mathbf{\Phi} \tag{2}$$

$$\forall e : \mathtt{Preterm}.\, (\cdot \vdash e : \mathtt{bool}) \longrightarrow (\cdot \vdash e \equiv \mathtt{true} : \mathtt{bool}) \vee (\cdot \vdash e \equiv \mathtt{false} : \mathtt{bool}) \tag{3}$$

Proof. First we prove that $\cdot \vdash \mathtt{true} \equiv \mathtt{false} : \mathtt{bool}$ does not hold in meta-space, *i.e.* assuming this equation we may deduce $\mathbf{\Phi} = \top$. By Lemma 24 we know that $[\![\mathtt{true}]\!]^{\mathsf{M}^*} = [\![\mathtt{false}]\!]^{\mathsf{M}^*}$ where M^* is our synthetic logical predicate; unfolding the interpretation function and the definition of M^*, we conclude $\eta_{\mathbf{\Phi}}.0 = \eta_{\mathbf{\Phi}}.1 : \mathbf{\Phi} \bullet \mathbf{2}$, which is equivalent to $\mathbf{\Phi} \bullet (0 = 1) = \mathbf{\Phi} \bullet \bot = \mathbf{\Phi}$.

Now we fix a preterm $e : \mathtt{Preterm}$ such that $\cdot \vdash e : \mathtt{bool}$ is derivable to prove that either $\cdot \vdash e \equiv \mathtt{true} : \mathtt{bool}$ or $\cdot \vdash e \equiv \mathtt{false} : \mathtt{bool}$ is derivable. By the ***synthetic adequacy postulate*** it suffices to check that either $[\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}} = [\![\mathtt{true}]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}}$ or $[\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}} = [\![\mathtt{false}]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}}$, *i.e.* that either $[\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}} = \eta_{\mathbf{\Phi}}.\mathsf{M}.\mathsf{true}$ or $[\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}} = \eta_{\mathbf{\Phi}}.\mathsf{M}.\mathsf{false}$. We consider the interpretation of e into the synthetic logical predicate, *i.e.* $[\![e]\!]^{\mathsf{M}^*} : \{\mathsf{M}^*.\mathsf{el}\, \mathsf{M}^*.\mathsf{bool} \mid \mathbf{\Phi} \hookrightarrow [\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}}\}$. We unfold the type of this element:

$$\{\mathsf{M}^*.\mathsf{el}\, \mathsf{M}^*.\mathsf{bool} \mid \mathbf{\Phi} \hookrightarrow [\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}}\} \;\cong\; \mathbf{\Phi} \bullet \{i : \mathbf{2} \mid \mathsf{enum}\, i = [\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}}\}$$

We may therefore suppose that we have an element $i : \mathbf{2}$ such that $\mathsf{enum}\,(\eta_{\mathbf{\Phi}}.i) = [\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}}$. If $i = 0$ then we have $[\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}} = \eta_{\mathbf{\Phi}}.\mathsf{M}.\mathsf{true}$, and if $i = 1$ we have $[\![e]\!]^{\mathbf{\Phi} \Longrightarrow \mathsf{M}} = \eta_{\mathbf{\Phi}}.\mathsf{M}.\mathsf{false}$. □

1.7 Validating the postulates of synthetic Tait computability

How do we connect our synthetic constructions to reality, and thus derive a canonicity result in ordinary mathematics? This is done by finding a *category* $\mathcal{E}$ that models all the postulates of synthetic Tait computability in such a way that the meta-space fragment of $\mathcal{E}$ is simply the category of sets, and thus any theorems proved about the "actual" syntax described in Section 1.6.1 will hold in the ordinary, set-theoretic sense.

It is unavoidable that it requires some knowledge of category theory to construct the category $\mathcal{E}$ and check that it satisfies the requisite properties. The purpose of these lectures is not to teach category theory, so for this section only we assume familiarity of certain basic constructions such as presheaves and comma categories; we also assume familiarity with the fact that dependent type theory can be interpreted into a topos (see Awodey et al. for an introduction), but we will not get specific enough for the details of this interpretation to matter.

1.7.1 The syntactic category of typed λ-calculus

We first describe the syntactic category $\mathbb{T}$ of typed λ-calculus; the objects of this category are *contexts* and the morphisms are definitional equivalence classes of simultaneous substitutions. In fact, we can re-use our synthetic construction of "actual" syntax from Section 1.6.1 by replaying it in ordinary set theory. This can be done by executing the definitions of `Ctx`, `Type`, `Preterm`, $\equiv_\alpha$, $(\Gamma \vdash e : \sigma)$ and $(\Gamma \vdash e \equiv e' : \sigma)$ in the language of set theory under the definitional extension $\mathbf{\Phi} := \bot$.

1.7.2 The object-space topos and the generic model of typed λ-calculus

Next we embed the syntactic category into a *topos* in order to facilitate dependent type theoretical language and logical reasoning; this topos is intended to interpret the ***object-space fragment*** of our synthetic language. The simplest topos embedding that we can consider is the ***Yoneda embedding*** $\mathsf{y}_{\mathbb{T}} : \mathbb{T} \hookrightarrow \mathbf{Pr}\,\mathbb{T}$ where $\mathbf{Pr}\,\mathbb{T}$ is the category of functors $\mathbb{T}^{\mathrm{op}} \longrightarrow \mathbf{Set}$ and $\mathbb{T}^{\mathrm{op}}$ is the ***opposite*** category of $\mathbb{T}$. The presheaf topos $\mathbf{Pr}\,\mathbb{T}$ interprets a model of intuitionistic dependent type theory with as many universes as exist in **Set**. Thus, synthetic object-space constructions can be replayed in $\mathbf{Pr}\,\mathbb{T}$ by simply setting $\mathbf{\Phi} := \top$; in particular, we have a type $\mathsf{STLC}_{\mathcal{U}}$ for each universe $\mathcal{U} \in \mathbf{Pr}\,\mathbb{T}$ classifying internal models of type λ-calculus.

Already in $\mathbf{Pr}\,\mathbb{T}$ we may define the generic model $\mathsf{M} : \mathsf{STLC}_{\mathcal{U}}$ where $\mathcal{U}$ is any universe. The presheaf of types M.type is given by the *constant* functor sending each context Γ to the set `Type` of analytic types. The dependent presheaf of elements M.el sends each context Γ and type $\sigma \in \mathtt{Type}$ to the set of morphisms $\hom_{\mathbb{T}}(\Gamma, (x : \sigma))$, *i.e.* the set of definitional equivalence classes of well-typed preterms $\Gamma \vdash e : \sigma$.

1.7.3 The global sections functor, or the *frontier* between object and meta

An equivalence class of *closed terms* of type σ is simply a morphism $\mathbf{1} \longrightarrow (x : \sigma)$ where $\mathbf{1}$ is the terminal object of $\mathbb{T}$, *i.e.* the empty context $\cdot$. Generalizing slightly, an equivalence class of *closed substitutions* into context Γ is a morphism $\mathbf{1} \longrightarrow \Gamma$. Generalizing a little more, we might say that a closed element of a presheaf $E \in \mathbf{Pr}\,\mathbb{T}$ is morphism $\mathbf{1} \longrightarrow E$, which is (by the Yoneda lemma) the same as an element of the set $E(1)$. It is clear that the set of closed elements of the representable presheaf $\mathsf{y}_{\mathbb{T}}\Delta$ is the same as the set of equivalence classes of closed substitutions into context Γ.

The functor that sends a presheaf to its set of closed elements is called the *global sections functor* $\mathbf{\Gamma} : \mathbf{Pr}\,\mathbb{T} \longrightarrow \mathbf{Set}$. This functor is important for us because it expresses the relationship between "object" and "meta" space; it will turn out that the global sections functor can be used to model the (synthetic) transition from an object-space type A to its meta-space type of closed elements $\mathbf{\Phi} \bullet (\mathbf{\Phi} \Longrightarrow A)$.

1.7.4 Gluing object and meta-space together

We can construct a new topos $\mathcal{E}$ that combines object-space and meta-space in a way that is determined by the global sections functor, reflecting the sense in which meta-space elements should be thought of as *closed elements*. There are several ways to construct this category which each have their own trade-offs, so we will choose the simplest and most intuitive construction.

An object of $\mathcal{E}$ is a pair of an object-space object $A \in \mathbf{Pr}\,\mathbb{T}$ together with a meta-space family of sets $(\bar{A}_x \in \mathbf{Set} \mid x \in \mathbf{\Gamma} A)$. A morphism $(A, \bar{A}) \longrightarrow (B, \bar{B})$ is defined to be a pair of a morphism $f : A \longrightarrow B \in \mathbf{Pr}\,\mathbb{T}$ together with a dependent function $\bar{f}$ of the following form in **Set**:

$$\bar{f} : \prod_{a \in \mathbf{\Gamma} A} A_a \longrightarrow B_{f(a)}$$

Remark 26. We can make an analogy to support the intuitions of those who are familiar with traditional logical predicates; considering the case where $A = \mathsf{y}_{\mathbb{T}}\sigma$, for any closed term $\cdot \vdash e : \sigma$, the set $\bar{\sigma}_e$ should be thought of as the collection of *proofs* that e is computable. A morphism from $(\sigma, \bar{\sigma})$ to $(\tau, \bar{\tau})$ then contains an actual equivalence class of terms $[x : \sigma \vdash e : \tau]$ together with a proof that e takes *computable* closed terms of type σ to *computable* closed terms of type τ.

The category $\mathcal{E}$ that we have constructed can be alternatively phrased in terms of the ***comma construction*** $\mathcal{E} = \mathbf{Set}\downarrow\Gamma$, also known as the ***Artin gluing*** of the global sections functor.The details of these constructions do not matter in the end, but they do inform the shape that various constructions will take. In the end, it is not difficult for an experienced user of category theory to show that these are all equivalent and that $\mathcal{E}$ is a Grothendieck topos, and thus models intuitionistic dependent type theory. All these facts can be black-boxed by the reader who is not comfortable with category theory.

We have an embedding of object-space into $\mathcal{E}$, *i.e.* a fully faithful functor $\mathcal{J} : \mathbf{Pr}\,\mathbb{T} \hookrightarrow \mathcal{E}$ that sends each presheaf $A \in \mathbf{Pr}\,\mathbb{T}$ to the family that asserts a trivial meta-space component over each closed element of A:

$$\mathcal{J}(A) := (A, \{a \longmapsto \mathbf{1}\})$$

We likewise have an embedding of meta-space into $\mathcal{E}$, *i.e.* a fully faithful functor $K : \mathbf{Set} \hookrightarrow \mathcal{E}$ sending each set $S \in \mathbf{Set}$ to the family that asserts a trivial object-space component with S in its unique fiber:

$$K(S) := (\mathbf{1}, \{* \longmapsto S\})$$

1.7.5 Validating the postulates

We will now substantiate our synthetic assumptions in the topos $\mathcal{E}$. The proposition $\mathbf{\Phi}$ is defined to be the pair $(\mathbf{1}, \{* \longmapsto \emptyset\}) \in \mathcal{E}$. It can be seen that $\mathbf{Pr}\,\mathbb{T}$ is canonically equivalent to the slice $\mathcal{E}\downarrow\mathbf{\Phi}$, and that under this identification the embedding $\mathcal{E}\downarrow\mathbf{\Phi} \simeq \mathbf{Pr}\,\mathbb{T} \hookrightarrow \mathcal{E}$ sends a dependent object $\mathbf{\Phi} \vdash A$ to its dependent product $\prod_{\mathbf{\Phi}} A$, which we have written $\mathbf{\Phi} \Longrightarrow A$. Thus under these identifications, the generic model $\mathsf{M} : \mathbf{\Phi} \Longrightarrow \mathsf{STLC}_{\mathcal{U}}$ can be interpreted to be exactly the generic model that we constructed in Section 1.7.2 using the actual syntax of typed λ-calculus. That the refinement types $[\mathbf{\Phi} \hookrightarrow x : A \mid Bx]$ can be modeled in $\mathcal{E}$ follows from results of Gratzer et al.

The last remaining thing to check is the ***synthetic adequacy postulate***, which is an immediate consequence the fact that both the Yoneda embedding and $\mathcal{J} : \mathbf{Pr}\,\mathbb{T} \hookrightarrow \mathcal{E}$ are fully faithful and preserve products and exponentials (function spaces).

Bibliography

[Awodey, Gambino, and Hazratpour, 2021] Steve Awodey, Nicola Gambino, and Sina Hazratpour. Kripke-Joyal forcing for type theory and uniform fibrations. Unpublished manuscript, 2021.

[Bizjak, Grathwohl, Clouston, Møgelberg, and Birkedal, 2016] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures: 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, pages 20–35, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49630-5. https://doi.org/10.1007/978-3-662-49630-5$_2$.

[Carette, Kiselyov, and Shan, 2007] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated. In Zhong Shao, editor, *Programming Languages and Systems*, pages 222–238, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-76637-7.

[Eilenberg and Steenrod, 1945] Samuel Eilenberg and Norman E. Steenrod. Axiomatic approach to homology theory. *Proceedings of the National Academy of Sciences*, 31 (4): 117–120, 1945. https://doi.org/10.1073/pnas.31.4.117.

[Fiore and Plotkin, 1996] Marcelo P. Fiore and Gordon D. Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In Dirk van Dalen and Marc Bezem, editors, *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*, volume 1258 of *Lecture Notes in Computer Science*, pages 129–149. Springer, 1996. https://doi.org/10.1007/3-540-63172-0_36.

[Gratzer, Shulman, and Sterling, 2022] Daniel Gratzer, Michael Shulman, and Jonathan Sterling. Strict universes for Grothendieck topoi. Unpublished manuscript, February 2022.

[Grothendieck, 1957] Alexander Grothendieck. Sur quelques points d'algèbre homologique, I. *Tôhoku Mathematical Journal*, 9 (2): 119–221, 1957. https://doi.org/10.2748/tmj/1178244839.

[Hoeven et al., 1998] J. van der Hoeven et al. GNU TeXmacs. https://www.texmacs.org, 1998.

[Hofmann, 1999] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 204–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0158-3. URL http://dl.acm.org/citation.cfm?id=788021.788940.

[Kock, 2006] Anders Kock. *Synthetic Differential Geometry*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2 edition, 2006. https://doi.org/10.1017/CBO9780511550812.

[Mac Lane and Moerdijk, 1992] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: a first introduction to topos theory*. Universitext. Springer, New York, 1992. ISBN 0-387-97710-4.

[Riehl and Shulman, 2017] Emily Riehl and Michael Shulman. A type theory for synthetic ∞-categories. *Higher Structures*, 1: 147–224, 2017. URL https://journals.mq.edu.au/index.php/higher_structures/article/view/36.

[Sterling, 2021] Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, 2021. CMU technical report CMU-CS-21-142.

[Sterling and Angiuli, 2021] Jonathan Sterling and Carlo Angiuli. Normalization for cubical type theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15, Los Alamitos, CA, USA, July 2021. IEEE Computer Society. https://doi.org/10.1109/LICS52264.2021.9470719.

[Sterling and Harper, 2021] Jonathan Sterling and Robert Harper. Logical relations as types: Proof-relevant parametricity for program modules. *Journal of the ACM*, 68 (6), October 2021. ISSN 0004-5411. https://doi.org/10.1145/3474834.

[Sterling and Harper, 2022] Jonathan Sterling and Robert Harper. Sheaf semantics of termination-insensitive noninterference. In Amy Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, August 2022. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.FSCD.2022.15.