URSS

Research Output

Introduction to Type Theory

Type Theory for Mathematics

2025-10-09

2111082

Contents

Foreword

1 Introduction

Sometimes, we see mathematical questions that don't appear "grammatically correct", so to speak. For instance,

- "Is [0,1] closed?"
- "Is $\mathbb{Z}$ a group?"
- "What is the fundamental group of $\mathbb{R} \sqcup S^1$?"
- "Is $\iota : \mathbb{Q} \hookrightarrow \mathbb{R}$ an epimorphism?"

Or for more exaggerated examples,

- "Is $\pi \in \log$?"
- "Is 3 surjective?"
- "Is $\sqrt{2}$ freely generated?"
- "What is the value of $\int \mathbb{Z}\, dx$?"

The problem here is that these questions have *type errors*. For instance, the "is X closed" predicate applies to a *pair* of *topological* spaces (A,S) with $S \subset A$, and not a lone *set* like [0,1], while "is X a group" applies to a pair $(G,*)$, consisting of a ground set G, and an operation $*$.

In the most commonly used foundations of mathematics – ZFC set theory, and more broadly, first-order logic – these grammatical quirks appear at an even more fundamental level. In most standard presentations, first-order logics are formulated in a *single-sorted* or *untyped* manner: when we write a variable x, we formally mean "some element of the underlying domain", without the possibility of ascribing any further structure or specification to x. All terms of first-order logic, regardless of how they are constructed, thus denote elements of this single domain, and all quantifiers range over the entire universe. Similarly, any function and relation symbols in our language are also taken to be total over the entire universe, so the different "kinds" of arguments and values of such symbols cannot be syntactically distinguished from one another.

Nothing in the formal system prevents us from writing expressions like $x + 1$ or $\mathsf{prime}(x)$ even if the formulae only make sense relative to certain intended interpretations of the variable x. For instance, if x denotes, say, a set, these mismatches are still syntactically valid even if they are nonsensical ("*ill-typed*"), since the underlying logical syntax does not distinguish between any of its objects. Whether we intend to quantify over numbers, sets, or other objects, they are all treated in this framework as belonging to the same undifferentiated universal soup.

ZFC inherits this agnosticism: its domain consists only of sets, and so all variables must range over all sets. This is especially visible when attempting to quantify over specific sets; we often write things like

$$\forall n \in \mathbb{N}. n + 1 > n$$

to express a property about the elements of the particular set $\mathbb{N}$. In doing so, we are really trying to express that the variable n should denote a natural number instead of ranging over all objects in the domain. However, the syntax of standard first-order logic as used in ZFC does not permit the restriction of quantifiers in this way, and so instead encodes it with an unrestricted quantifier and an auxiliary predicate acting as a guard:

$$\forall x. x \in \mathbb{N} \to (x + 1 > x)$$

Similarly, the relation symbol $\in$ is also necessarily compatible with any two objects of the domain, so expressions like "$x \in y$" are always valid, regardless of what x and y are meant to represent. From the point of view of ZFC, the sentence "is $\pi \in \log$?" is a perfectly legitimate question, with an unambiguous (if utterly uninformative) answer: it is false*, because the set we use to represent π does not happen to be an element of the set we use to represent log.

The benefit of this style of axiomatisation is simplicity – by only allowing a single undifferentiated domain, one avoids the need to introduce and track multiple kinds of objects, which keeps the syntax uniform and the semantics comparatively straightforward. This uniformity also allows for an elegant and minimalistic foundation where a small number of rules suffice to encode the vast majority of modern mathematics. Clearly, this works on a technical level – ZFC is the most popular foundation of mathematics for a reason – but this simplicity comes at the cost of constantly having to simulate "grammatical correctness" by indirect means; or not at all (i.e. "is $\pi \in \log$?"), instead judging whether a statement is mathematically meaningful *externally.*

1.1 Typing as Annotation

One way to address the grammatical permissiveness of ordinary first-order logic is to enrich the formal language with *types* (often called *sorts* instead, to avoid conflicts with the main subject of this paper). Instead of having a single undifferentiated domain of discourse, we allow multiple domains, each associated with a different kind of object. Variables, functions, and predicates are then annotated with their types, and expressions are only well-formed if they respect these annotations.

For instance, we might specify that addition has type

$$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

so that an expression like $x + 1$ is well-formed only when x is of type $\mathbb{N}$; or that the membership relation has type

$$\in : \mathcal{U} \times \mathcal{U} \to \mathsf{Prop}$$

where $\mathcal{U}$ denotes the universe of sets and Prop the universe of logical propositions.

By strictly enforcing these annotations, the system can rule out ill-typed expressions such as "$\pi \in \log$" on purely syntactic grounds; such malformed expressions are not merely assigned a worthless truth value, but are made grammatically impossible to form in the first place.

This idea of a type annotation is already familiar in mathematics. As mentioned previously, we often informally write things like "$\forall x \in X, \varphi(x)$" to express a property φ about the elements of a particular set X – in this notation, the membership attached to the quantifier is semantically intended to be a type annotation, restricting the type of the variable x. While in standard first-order logic, this construction is forced to be encoded with a guard and implication, typed first-order logic allows us to collapse this distinction and make the intended semantics explicit in the syntax itself.

A crucial feature of this type annotation system is that, once the types of a few primitive operations are fixed, the types of more complex expressions can often also be deduced entirely mechanically. For example, from the declaration $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, we can immediately determine that in the expression $x + 1$, the variable x must have type $\mathbb{N}$, even if this was not stated explicitly. This process of deducing the types of subexpressions from their context is called *type inference.* This process is well-studied, and we have efficient algorithms for type inference in many systems of interest. In practice, this means that type annotations do not need to be written everywhere: once a few basic operation types are specified, the rest of the grammar enforces itself.

Formally, these systems are well studied: *many-sorted logic* generalises first-order logic to multiple domains of discourse, while Church's *typed λ-calculus* extends this idea to functions, introducing a calculus of terms where each expression carries an explicit type. Importantly, these systems are conservative over their untyped counterparts: typing does not introduce new theorems about the underlying mathematics, it merely enforces a more disciplined grammar for writing them.

*Assuming the standard set-theoretic encoding of functions as ordered pairs of inputs and outputs.

1.2 Intrinsic and Extrinsic Typing

Up to this point, we have been speaking of “typing” in the sense of taking familiar mathematical objects and assigning them to different syntactic categories to rule out (some) meaningless expressions.

This much could be carried out on top of any existing foundation, including set theory. For instance, one could perform mathematics in ZFC, but then retroactively impose a typing discipline on top, carefully annotating every object with its intended type and only allowing constructions that respect these annotations.

This is what we have discussed so far, and it can be very useful to organise mathematics that is already phrased in set-theoretic terms in this way (and is what most mathematicians implicitly do), but the types in this style of typing are *extrinsic* to the logical system. Adding types in this way does not change the underlying ontology of mathematics: terms of this new typed theory are still the same raw untyped entities of the underlying system, and the typing rules only classify which terms are admissible. The types are thus necessarily external to the underlying system.

One major difficulty with this extrinsic typing arises from this disconnect between types and terms. Because the terms of the underlying system are fundamentally untyped, type information exists entirely independently, as a separate layer of annotation. Consequently, there is no guarantee that constructions will respect the intended semantics of a term; every expression must be checked explicitly against the typing rules. So, for instance, you must prove separately that an expression is well-typed before it corresponds to a valid mathematical construction.

This can be mitigated to a certain extent by type inference. However, this separation also complicates type inference, since the same raw term may correspond to many potential types, and any type inference algorithm must search through these possibilities without any guidance from the term’s structure. For example, the set $\{\varnothing,\{\varnothing\}\}$ is simultaneously:

- the von-Neumann ordinal 2;
- a relation R on a singleton set $\{\varnothing\}$;
- the Kuratowski ordered pair $\langle\varnothing,\{\varnothing\}\rangle$;
- and the unique topology on the singleton space $\{\varnothing\}$;

while the set $\{\varnothing,\{\varnothing\},\{\varnothing,\{\varnothing\}\}\}$ is simultaneously:

- the von-Neumann ordinal 3;
- the free arrow graph $\left[\varnothing \to \{\varnothing\}\right]$;
- and the Sierpiński topology on the two-point space $\{\varnothing,\{\varnothing\}\}$.

None of the intended semantics are apparent – or even possibly extractible in any sense – from these raw encodings. As can be seen, encoding complex objects in an untyped system often requires these kinds of *ad hoc* constructions, which further obscures the relationship between the term and its intended type.

The alternative is to build types *intrinsically* into the foundational grammar of the system itself, so that terms and their types are defined together, rather than sequentially. In such a framework, there is no stage at which one first produces untyped expressions and then checks afterwards whether they can be assigned types; instead, every well-formed expression arises already equipped with its type.

Apart from significantly simplifying type inference by allowing terms to structurally constrain their types, adopting an intrinsic approach to typing also shifts the way we think about mathematical objects. In an extrinsic setting, such as set theory, new notions are usually defined by reducing them to more primitive constructions: ordered pairs to particular sets, numbers to particular ordinals, and so on, and the properties we want the objects to exhibit are then proved as theorems about our choice of encodings. This makes it awkward to describe the *behaviour* of mathematical constructions directly, since every account must be mediated through their encodings.

In contrast, in an intrinsic setting, this operational viewpoint is especially natural: well-formed expressions can only ever exist *together* with their intended modes of use. One cannot first construct an object and then later ask how it should behave; instead, the type information of an object determines its behaviour from the outset. This perspective shifts attention away from what an object actually *is*, and instead towards how it *behaves*.

1.3 Typing as Specification

Consider the notion of an *ordered pair*. In standard set theories, this is typically defined via the Kuratowski construction:

$$\langle a,b\rangle \coloneqq \big\{\{a\},\{a,b\}\big\}.$$

and the corresponding projections can be defined as:

$$\pi_1\langle a,b\rangle = \bigcap\langle a,b\rangle$$
$$\pi_2\langle a,b\rangle = \bigcup\langle a,b\rangle \setminus \bigcap\langle a,b\rangle$$

In practice, mathematicians usually do not delve into these intricacies, because they are irrelevant *implementation details*. What actually matters about an ordered pair are its observable properties:

(i) Given two objects a and b, we can form their pair $\langle a,b\rangle$;

(ii) Given a pair $\langle a,b\rangle$, we can extract each component, a and b.

In other words, what we really mean by "ordered pair" is specified not by its reduction to more primitive notions – set-theoretic or otherwise – but by the fact that it supports exactly these two basic operations. An ordered pair is not a particular set, but a mathematical object governed by a simple *interface*: it can be built from two components, and the two components can be recovered.

If we abstract away these implementation details, we could instead view an ordered pair precisely as being *defined* by these properties, as an abstract interface. More generally, *every* mathematical object can be defined in this way, through their abstract interfaces.

By focussing on the operations that define an object, rather than its particular material representation, we can reason about mathematical structures without needing to know about any underlying encoding in the background.

Consequently, the particular *instances* of a mathematical structure are of secondary importance. What actually matters are the operations they support and the relations they satisfy – but all of this information is captured at the level of interfaces, rather than instances; it is the parent interface that determines how an object can be used and combined – or reasoned about. Reasoning in mathematics, then, becomes reasoning about these interfaces, rather than about any individual representative element.

Perhaps surprisingly, rather than merely not needing to know the details about the underlying encoding of an interface, it turns out that we do not need any underlying encoding in the first place; these abstract interfaces – *types* – are sufficient to organize and develop mathematics on their own.

The study of these pure "unimplemented" interfaces, is then called *type theory*.

2 Simple Type Theory

When ZFC is presented as a foundation for mathematics, it is explicitly formulated as a theory *within* first-order logic. The *deductive* layer – the syntax and inference rules of first-order logic – comes first, and then the axioms of set theory are added on top. Thus, ZFC has a "two-layer structure: first-order logic provides the underlying deductive calculus, while ZFC set theory is just a particular first-order theory stated within that calculus. Moreover, this separation is strict: nothing about the logical framework depends on the choice of axioms, and one could replace the set-theoretic axioms with any other collection of first-order sentences without altering the deductive system itself.

In contrast, type theory does not presuppose any underlying logical system at all. It is a self-contained logical calculus in its own right, where both construction and *reasoning* proceed entirely in terms of types and their relations.

2.1 Judgements

In first-order logic, the basic objects are *formulae*, and we ask questions such as whether a formula is *well-formed*, *true* in a structure $\mathcal{M}$, or *provable* from a set of assumptions Γ:

$$\varphi \text{ is a well-formed formula} \qquad \mathcal{M} \vDash \varphi \qquad \Gamma \vdash \varphi$$

These statements are not themselves formulae of the logic, but are instead external metatheoretic assertions *about* formulae, called *judgements*, and the *inference rules* of the theory describe how new judgements may be derived from existing ones.

In type theory, the basic objects are not formulae, but types; consequently, the judgement forms are also different. Unlike ZFC set theory, type theory is, by default, a purely syntactic framework. Its primitive judgements are not statements about whether formulae are true or false in some semantic structure, but are rather assertions concerning the *well-formedness* of expressions.

In this sense, every statement in type theory is ultimately a claim of well-formedness – that a particular symbol, expression, or collection of expressions can appear in the language of the theory – and the inference rules, rather than describing logical entailment between propositions, prescribe how complex expressions may be constructed and when they are considered equivalent.

The six fundamental judgement forms of our initial simple type theory are as follows:

$$\begin{array}{ccc} \Gamma \ \mathsf{ctx} & A \ \mathsf{type} & \Gamma \vdash a : A \\ \Gamma \equiv \Delta \ \mathsf{ctx} & A \equiv B \ \mathsf{type} & \Gamma \vdash a \equiv b : A \end{array}$$

Every well-formed expression in type theory is introduced and manipulated solely through the derivability of these judgements.

We will only focus on a few of these for now, as most of these are trivial or uninteresting for simple types, and are only there as formal boilerplate.

The simplest form of judgement is the *typing judgement*:

$$x : A$$

which is read as "x is a *term* of *type* A".

This judgement form bears some superficial resemblance to the set-theoretic proposition

$$x \in A$$

but the two are fundamentally different kinds of statements. Membership is a relation *between* objects, while typing is a judgement that *introduces* objects.

In set theory, $x \in A$ is a proposition *within* the theory, concerning two *pre-existing* objects x and A. The universe of sets is taken to already contain both x and A, regardless of any prior relation between them, and the membership relation can then be applied afterwards to produce a proposition which may turn out to be true or false.

In contrast, the typing judgement $x : A$ is a *metatheoretic* assertion that *defines* the symbol x in the first place. These typing judgements are the basic means by which symbols are brought into the theory: always together with a type. A term cannot exist independently of its type, so to write $x : A$ is not to assert a fact about two already-given objects, but rather to *declare* that x is a term of type A in the first place. Consequently, the statement $x : A$ is not something one can later *prove* about a new symbol x.

In this sense, the typing judgement $x : A$ is much more similar to the first-order logic judgement "x is a variable" (as opposed to a function symbol or relation symbol, etc.). One cannot, for instance, take an arbitrary symbol x and then *prove* that it is a variable; rather, the grammar of first-order logic stipulates from the outset that certain symbols just *are* variables.

Beyond this declarative use in introducing new symbols, the typing judgement also serves a crucial operational function in type theory. Once a few basic symbols and constructions have been introduced, further typing judgements may appear as *derived* rather than declared, propagating and transforming type information through derivations. For example, if we have a function $f : A \to B$ and a value $a : A$, then we may deduce that $f(a) : B$. In this case, the judgement does not introduce a new symbol, but is rather expressing that a certain expression has a particular type according to the inference rules of the theory. This inferential use of typing underlies the deductive and computational aspects of type theory.

However, just as terms are introduced and reasoned about through typing judgements, the formation of types themselves must also be governed by explicit rules. Before we can assert that some symbol x has type A, we must first know that A is a valid type. This requires separate form of judgement, called a *type formation*:

$$A \ \mathsf{type}$$

This asserts that a given expression denotes a *well-formed* type. The judgement A type plays the same grammatical role for types that $x : A$ does for terms, just at a "higher level".

Our theory will include an infinite collection of *base* or *atomic* types A,B,C,… that always satisfy this judgement without any assumptions. These primitive types are given only *opaquely*: we know they exist, and we can declare terms to inhabit them in assumptions, but their internal structure is unspecified. Later, we will introduce more complex types that will be formed from these base types, as well as other non-opaque base types. For now, the theory treats each of these types as an indivisible symbol whose sole role is to serve as a target for typing declarations.

Having introduced the judgements for forming both terms and types, we can now begin to consider how these elements interact. To form a term like $x + 1$, the term x must itself already have a type for the expression to make sense. Since every term must be introduced together with its type, it is not enough to consider isolated typing judgements to reason about such terms.

To keep track of the assumptions under which variables are meaningful and to manage how terms depend on these variables, we need some bookkeeping to keep track of assumptions when reasoning about terms that depend on variables.

A *context* Γ is a list of typing judgements recording which terms are available and what types they belong to:

$$\Gamma \ = \ x_1 : A_1, x_2 : A_2, \dots, x_n : A_n$$

The statement that a list Γ is a valid context is also a judgement, written as

$$\Gamma \ \mathsf{ctx}$$

This judgement is governed by the following inference rule:

$$\frac{}{\cdot \ \mathsf{ctx}} \ (\text{empty}) \qquad\qquad \frac{\Gamma \ \mathsf{ctx} \quad \Gamma \vdash A \ \mathsf{type}}{\Gamma, x : A \ \mathsf{ctx}} \ (\text{ext})$$

That is, the empty list of assumptions is always a context; and given a context, extending it by a typing judgement (x must be fresh here) yields another context. Since contexts are lists, this generates all possible contexts, as required.

The details of context formation are not particularly significant for simple types, so we will postpone a detailed analysis to a later section, when we have a more expressive type theory. We will also suppress context formation judgements from all inference rules from now on, as what it means to be a context for simple types is fairly straightforward, and we will assume that Γ, Δ, etc. always denote arbitrary valid contexts.

With a context in place, the general typing judgement takes the form of a *sequent*:

$$\Gamma \vdash t : A$$

which is read as, "the term t has type A *in context* Γ." If Γ is empty, then we will return to writing the simpler typing judgement $x : A$ instead of $\cdot \vdash x : A$.

It is important to note that contexts are purely syntactic devices: they record *assumptions* about terms, rather than *existence*. That is, a judgement

$$x : A, y : B \vdash t : C$$

is not an existential assertion about x or y, but a conditional one: it expresses that the term t would be well-typed under the hypothetical provision of x and y as terms of the indicated types. This is directly analogous to reasoning with free variables in first-order logic.

It is also possible to have seemingly distinct contexts that describe equivalent situations. For example, the contexts

$$\Gamma \equiv x : A, y : B \qquad \text{and} \qquad \Delta \equiv a : A, b : B$$

are syntactically distinct, but really represent the same collection of assumptions, since the choice of symbol used to denote something shouldn't meaningfully affect our theory.

Two contexts Γ and Δ are *α-equivalent* if there is an order and type preserving bijection between the variable names of the two contexts, such that all terms and judgements formed relative to Γ can be consistently renamed according to that bijection. In other words, the identity of a context does not depend on the specific choice of variable names, but only on its structure as a typing environment. Replacing a term in a context by another fresh name of the same type in this way is called *α-conversion*.

Note that, despite the fact that we are renaming "*free*" variables, this is really the same thing as α-conversion in the λ-calculus (where *bound* variables are renamed), because the typing declaration $t : A$ acts as a binder for all subsequent occurrences of x in the sequent. So, while the variables declared in a context Γ are syntactically *free* in the term $t : A$, they are *bound* in the overall sequent $\Gamma \vdash t : A$. We can really think of λ-abstraction as a kind of *local* binder, while the sequent $\Gamma \vdash t : A$ is a *global* binder, and α-equivalence applies uniformly – just to different scopes.

This identification of expressions differing only by consistent renaming of bound variables is an essential part of the syntactic character of type theory – it ensures that variable names carry no semantic content and prevents distinctions between syntactically different but structurally identical judgements.

Thus, two contexts (or terms written within them) that differ only by reordering or by systematic renaming of bound variables are treated as equivalent for all formal purposes. To make such equivalences precise, we must extend our system to include explicit judgements for equality between syntactic entities; between types, terms, and even contexts themselves.

2.1.1 Judgemental Equality

Alongside typing judgements, the other basic kind of assertion about terms is the *term equality* judgement:

$$t \equiv u : A$$

read as "t and u are *definitionally* equal terms of type A." If the type A is obvious, it can be omitted, and we just write $t \equiv u$.

It is important to distinguish this form of equality from equality as it appears in set theory or first-order logic. In set theory, equality is a binary relation between objects inside the theory. The *proposition* $t = u$ is something that can be true or false and can be reasoned about.

In contrast, the type-theoretic judgement $t \equiv u : A$ is not a proposition, but again, a metatheoretic assertion about the syntactic behaviour of terms: it says that t and u are indistinguishable for all purposes of computation and deduction within the type system. This judgement captures what is sometimes called *definitional* or *judgemental* equality, as opposed to the familiar *propositional* equality (which we will define later on in a more expressive type theory).

For example, if we have a natural-valued function $f :\equiv \lambda x.x + 1$ being applied to an argument 2, then the expressions

$$f(2), \qquad (\lambda x.x + 1)2, \qquad 2 + 1$$

should intuitively be judgementally equal: the first two are equal *by the definition of* f, while the last two are equal *by* β*-reduction*, i.e. the definition of function application.

Moreover, we will define natural addition via the Peano axioms as:

$$n + 0 \equiv n, \qquad n + \mathsf{succ}(m) \equiv \mathsf{succ}(n + m)$$

so $2 + 1$ is also judgementally equal to 3:

$$\begin{aligned} 2 + 1 &\equiv \mathsf{succ}(\mathsf{succ}(0)) + \mathsf{succ}(0) \\ &\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0)) + 0) \\ &\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0))) \\ &\equiv 3 \end{aligned}$$

In each case, the equality is *syntactic* or *computational*, and is not something that requires constructing any kind of proof term. Intuitively, judgemental equality captures of all the "obvious" kinds of equalities that can be obtained by unfolding definitions or mechanically applying computation rules.

There is also a notion of judgemental equality for types

$$A \equiv B \ \mathsf{type}$$

and also for contexts:

$$\Gamma \equiv \Delta \ \mathsf{ctx}$$

All three forms of equality judgement are equipped with inference rules that make them equivalence relations on their own sorts of objects. For instance, for terms, we have the inference rules:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A}(\text{refl}) \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A}(\text{sym}) \qquad \frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}(\text{trans})$$

and analogous rules hold for types and contexts, replacing "$- : A$" with "$-$ type" and "$-$ ctx", respectively.

All three are also stable under α-conversion – that is, renaming of bound variables does not affect the validity of any equality judgement. And finally, all three forms of equality judgement also interact coherently: each is stable under substitution of judgementally equal objects of the other sorts. For instance, if $\Gamma \vdash a \equiv b : A$ and $\Gamma \equiv \Delta$ ctx, then $\Delta \vdash a \equiv b : A$:

$$\frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \equiv \Delta \ \mathsf{ctx}}{\Delta \vdash a \equiv b : A}$$

and similar for all other possible pairings.

For simple types, equality judgements for contexts and types are trivial, and we will again suppress them in the following.

2.2 Types

So far, we have postulated the existence of an infinite collection of base types, but we can also construct new types by applying *type formers* to existing to obtain *compound types* with more structure. Compound types are not primitive notions like atomic types, but are defined together with additional functions and rules describing how their terms behave. These rules typically consist of:

- *introduction rules* – ways to build terms of the type using *constructors*;
- *elimination rules* – ways to use terms of the type using *destructors*;
- *computation* or *β-rules* – how a destructor applied to a constructor (a *redex*) reduces to a canonical form;
- and *uniqueness* or *η-rules* – ways that identify when two terms of the type are equal.

In our simple type theory, we have only three type formation rules: given types A and B, we may form:

- the *function type* $A \to B$;
- the *product type* $A \times B$;
- and the *sum type* $A + B$.

Formally, in terms of judgements:

$$\frac{A \ \mathsf{type} \qquad B \ \mathsf{type}}{A \to B \ \mathsf{type}} \qquad \frac{A \ \mathsf{type} \qquad B \ \mathsf{type}}{A \times B \ \mathsf{type}} \qquad \frac{A \ \mathsf{type} \qquad B \ \mathsf{type}}{A + B \ \mathsf{type}}$$

We now give the rules for each compound type.

2.2.1 Function Types

- Introduction:
$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}(\to_{\mathrm{I}})$$
Given a term $t : B$ depending on free variable $x : A$, we may form a term of type $A \to B$, which we may think of as a function taking x as argument.
- Elimination:
$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}(\to_{\mathrm{E}})$$
- Computation:
$$\frac{}{\Gamma \vdash (\lambda x.t)a \equiv t[a/x]}(\to_{\beta})$$
If we have a function $f : A \to B$ and an argument $a : A$, then the application $f(a)$ is of type B. Moreover, applying a function abstraction to an argument reduces to substitution of the argument for the bound variable in the body.
- Uniqueness:
$$\frac{}{\Gamma \vdash f \equiv \lambda x.f(x)}(\to_{\eta})$$
Every function f is judgementally equal to the abstraction that maps x to $f(x)$. That is, a function is determined uniquely by its action on arguments. This extensionality principle is also called *η-conversion.*

2.2.2 Product Types

- Introduction: given terms $a : A$ and $b : B$, we can obtain a term $\langle a,b\rangle$ of type $A \times B$.
$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a,b\rangle : A \times B}(\times_{\mathrm{I}})$$
- Elimination: from a term of a product type, we can obtain terms of each component type by projection:
$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{fst}(p) : A}(\times_{\mathrm{E}_1}) \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{snd}(p) : B}(\times_{\mathrm{E}_2})$$

- Computation:

$$\frac{}{\Gamma \vdash \mathsf{fst}\langle a,b\rangle \equiv a}(\times_{\beta_1}) \qquad \frac{}{\Gamma \vdash \mathsf{snd}\langle a,b\rangle \equiv b}(\times_{\beta_2})$$

  In particular, the destructors recover exactly the components we started with.

- Uniqueness: every pair is definitionally equal to one built from its projections: there are no "non-standard" terms that don't arise from pairing:

$$\frac{}{p \equiv \big\langle \mathsf{fst}(p),\mathsf{snd}(p)\big\rangle}(\times_{\eta})$$

2.2.3 Sum Types

- Introduction:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}(a) : A + B}(+_{\mathrm{I}_1}) \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}(b) : A + B}(+_{\mathrm{I}_2})$$

  to construct an element of a sum type, it suffices to provide a term from one of the components, and the two constructors $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ then keep track of which was the case.

- Destructors:

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma \vdash f : A \to C \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathsf{case}(t; f,g) : C}(+_{\mathrm{E}})$$

  The destructor $\mathsf{case}(-;-,-) : (A + B) \to (A \to C) \to (B \to C) \to C$ expresses that, to produce a term of type C from a term of type $A + B$, it suffices to give its behaviour on each summand separately.

- Computation:

$$\frac{}{\Gamma \vdash \mathsf{case}(\mathsf{inl}(a); f,g) \equiv f(a)}(+_{\beta_1}) \qquad \frac{}{\Gamma \vdash \mathsf{case}(\mathsf{inr}(b); f,g) \equiv g(b)}(+_{\beta_2})$$

  In particular, the destructor pattern matches $t : A + B$ against the two cases: $t \equiv \mathsf{inl}(a)$ for some $a : A$ or $t \equiv \mathsf{inr}(b)$ for some $b : B$, before applying the appropriate function.

- Uniqueness: any term of type $A + B$ is fully determined by how it behaves under case analysis.

$$\frac{}{\Gamma \vdash s \equiv \mathsf{case}\big(s; \lambda a.\mathsf{inl}(a),\lambda b.\mathsf{inr}(b)\big)}(+_{\eta})$$

  Again, this means that there are no "non-standard" terms that don't arise from injections.

2.2.4 Uniqueness and Syntactic Extensionality

In set-theoretic treatments of ordered pairs, one must separately verify that the chosen encoding satisfies a suitable extensionality principle, ensuring that equality of ordered pairs coincides with componentwise equality:

$$\langle a,b\rangle = \langle c,d\rangle \iff a = c \wedge b = d$$

Only once this property has been established can the projection functions then be recovered as derived operations. In this sense, the set-theoretic treatment fixes equality first, and then recovers the projections afterwards.

In contrast, the type-theoretic approach reverses this order of priority. Here, the projections are taken as primitive, and pair extensionality follows automatically as a syntactic consequence of this definition. Rather than equality collisions being a concern as when encoding these as sets, two pairs in type theory are equal if and only if their components are definitionally equal, since the only way a term of a product type can arise is via pairing. Thus, $\langle a,b\rangle \equiv \langle c,d\rangle$ if and only if $a \equiv c$ and $b \equiv d$, *by definition*.

This phenomenon – that the definitional equality of a type is determined entirely by the canonical forms allowed by its constructors – is a much more general pattern in type theory. For any type, the rules specifying how terms are formed and deconstructed automatically and fully determine when two terms are equal, giving a syntactic notion of extensionality that requires no reference to underlying representations.

2.3 Inference Rules

Now, just as inference rules in first-order logic allow us to derive new *formulae* from existing assumptions, type theory is also equipped with a collection of inference rules that allow us to derive new *typing sequents* from existing ones.

Simple type theory only has one primitive inference rule:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A} \text{(axiom)}$$

This is the *axiom* or *variable* rule: any variable declared in the context is a valid term of its declared type. This seems very simple, but in fact, we can prove quite a lot with this already.

An inference rule is *admissible* for a deductive system if whenever the premises of the rule are derivable in that system, then so is the conclusion. Equivalently, adding an admissible rule as primitive would not allow us to derive any new judgements that were not already derivable without it – any derivation using an admissible rule can be reconstructed without it.

2.3.1 Structural Rules

We can already prove quite a few *structural rules* that allow us to manipulate contexts more freely.

The *weakening rule* says that we can weaken a derivation by introducing more assumptions than we need. That is, if $\Gamma \vdash x : A$, and y is a fresh variable, then $\Gamma, y : B \vdash x : A$:

Theorem 2.1 (Weakening)**.** *The weakening rule*

$$\frac{\Gamma, \Delta \vdash x : A}{\Gamma, y : B, \Delta \vdash x : A} \text{(wk)}$$

is admissible, for y fresh in Γ.

Proof. We induct on the derivation of $\Gamma \vdash x : A$.

In every case, we can choose y fresh; if any binder in the derivation conflicts with y, we can first α-convert that binder so that it does not.

- If the last rule is axiom, the derivation ends with

  $$\frac{}{\Gamma, x : A, \Delta \vdash x : A} \text{(ax)}$$

  Since y is fresh, we may insert $y : B$ anywhere in the context; in particular,

  $$\frac{}{\Gamma, x : A, y : B, \Delta \vdash x : A}$$

  is still an instance of axiom. So weakening holds in this case.

- Supppose the last rule is abstraction (so $A \equiv C \to D$):

  $$\frac{\Gamma, z : C \vdash t : D}{\Gamma \vdash \lambda z.t : C \to D} (\to_{\mathrm{E}})$$

  where z is fresh for this subderivation. By applying the induction hypothesis to the premise, we have:

  $$\Gamma, y : B, \Delta, x : C \vdash t : D$$

  and abstraction again yields

  $$\Gamma, y : B, \Delta \vdash \lambda x.t : C \to D$$

  as required.

- Suppose the last rule is application:

$$\frac{\Gamma \vdash t_1 : C \to A \qquad \Gamma \vdash t_2 : C}{\Gamma \vdash t_1 t_2 : A}(\to_{\mathrm{E}})$$

  By the induction hypothesis, we have:

$$\Gamma, y : B \vdash t_1 : C \to A \qquad \text{and} \qquad \Gamma, y : B \vdash t_2 : C$$

  so by applying the application rule to these new judgements, we have

$$\Gamma, y : B \vdash t_1 t_2 : A$$

  as required.

- Product/sum/unit/void introduction and elimination rules follow identically: weaken each premise by the induction hypothesis, then reapply the rule in the new context. If any rule binds a variable, α-convert to keep y fresh.

■

All other proofs for structural rules proceed similarly, by induction on derivations, so we will omit the details from now on.

The *exchange rule* says that we may freely permute adjacent typing declarations.

Theorem 2.2. *The exchange rule*

$$\frac{\Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, \Delta \vdash t : C}(\mathrm{wk})$$

is admissible.

Proof. By induction on derivations. ■

Corollary 2.2.1. *Contexts may freely permute all typing declarations.*

Proof. Every permutation is generated by transpositions. ■

The *contraction rule* says that if two assumptions have the same type, then they may be merged, uniformly replacing all instances of one term with the other.

Theorem 2.3. *The contraction rule*

$$\frac{\Gamma, x : A, x' : A, \Delta \vdash t : B}{\Gamma, x : A, \Delta[x/x'] \vdash t[x/x'] : B}(\mathrm{contr})$$

is admissible.

The *substitution rule*

Theorem 2.4. *The substitution rule*

$$\frac{\Gamma, x : A, x' : A, \Delta \vdash t : B}{\Gamma, x : A, \Delta[x/x'] \vdash t[x/x'] : B}(\mathrm{subst})$$

is admissible.

$$\frac{\Gamma, x : A, y : A, \Delta \vdash t : B}{\Gamma, x : A, \Delta \vdash t[x/y] : B}\text{(substitution)}$$

$$\frac{\Gamma, x : A, x : A, \Delta \vdash y : B}{\Gamma, x : A, \Delta \vdash y : B}\text{(contraction)}$$

$$\frac{\Gamma \vdash x : A \qquad \Delta, x : A \vdash y : B}{\Gamma, \Delta \vdash y : B}\text{(cut)}$$

2.3.2 More Base Types

We can also introduce non-opaque base types in the same way. We introduce three additional base types as follows:

- the unit type;
- the void type;
- and the nat type.

The unit type is the type with a single inhabitant, denoted by $\star : \mathsf{unit}$, or in computer science contexts, by $() : \mathsf{unit}$. Its computation and uniqueness rules formalise the idea that there is essentially no information contained in values of this type, since they must necessarily be $\star$.

- Introduction:

$$\frac{}{\Gamma \vdash t : \mathsf{unit}}$$

  That is, we can always obtain a term t of type unit, without any assumptions.

- Elimination: to define a function $\mathsf{unit} \to B$, it suffices to provide a canonical representative $b : B$. Intuitively, since unit has exactly one inhabitant, there is no information in t to influence the choice of result; the term b is therefore the only output that can arise from eliminating t:

$$\frac{\Gamma \vdash t : \mathsf{unit} \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{elim}(b,t) : B}$$

- Computation: the computation rule formalises this expected behaviour; when applied to the canonical inhabitant $\star$, we obtain exactly the term x provided:

$$\mathsf{elim}(b,\star) \equiv b$$

- Uniqueness: since unit only has one term, every term of type unit is definitionally equal to to the canonical inhabitant $\star$:

$$t \equiv \star$$

The void type is the type with no inhabitants, reflected in the absence of any constructor – there should be no way to poduce a term of type void.

- Introduction: there is no constructor for the void type.
- Elimination:

$$\frac{\Gamma \vdash t : \mathsf{void}}{\Gamma \vdash \mathsf{abort}(t) : B}$$

  The destructor $\mathsf{abort} : \mathsf{void} \to B$ can be understood as a way of detecting inconsistency: if we can somehow derive a term of type void – which has no constructors – then our assumptions must be inconsistent and we should be able to obtain a term of any type we like.

- Computation: there are no constructors, and hence no redexes to compute.
- Uniqueness: there are no terms, and hence every term of void is trivially canonical.

The nat type represents the natural numbers, and can be defined from a base constant 0, and a successor function $\mathsf{succ} : \mathsf{nat} \to \mathsf{nat}$:

- Introduction:

$$\frac{}{\Gamma \vdash 0 : \mathsf{nat}} \qquad \frac{\Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(n) : \mathsf{nat}}$$

- Elimination: to eliminate a term of type nat, it suffics to provide an initial value $b : B$ and a recursion function $r : B \to B$ for successors. Then, for any $n : \mathsf{nat}$, we can form the recursion:

$$\frac{\Gamma \vdash b : B \quad \Gamma \vdash r : B \to B \quad \Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \mathsf{natrec}(b,r)(n) : B}$$

- Computation:

$$\mathsf{natrec}(b,r)(0) \equiv b$$
$$\mathsf{natrec}(b,r)(\mathsf{succ}(n)) \equiv r(\mathsf{natrec}(b,r)(n))$$

  For the base case 0, recursion returns the base case b, and for $\mathsf{succ}(n)$, recursion applies r to the recursive result on r. This is precisely the ordinary definition of simple recursion, as in set theory or category theory.

- Uniqueness: every term of type nat is definitionally equal to either 0 or $\mathsf{succ}(n)$ for some $n : \mathsf{nat}$. In particular, $0 \not\equiv \mathsf{succ}(m)$ for all $m : \mathsf{nat}$, and succ is injective.

2.4 Inductive Types

At this point, we should pause and extract some general patterns from the previous presentation nat type. The natural numbers are the prototypical example of an object that is *freely generated* by a collection of constructors: every term of type nat is produced by finitely many applications of those constructors, and eliminating a term of type nat is done by recursively handling each of the constructors.

Conversely, the uniqueness results about nat – the injectivity of succ, the disjointness of 0 and successors, and the statement that every term is built from the constructors – are not independent axioms, but *syntactic consequences* of the inductive definition itself.

We now give a treatment of more general types that are constructed in the same way as nat, namely, as minimal types closed under a finite collection of constructors.

An *inductive type* T is defined by a finite (say k-many) collection of *constructors* c_i, each of arity n_i:

$$c_i : A_{i,1} \to A_{i,2} \to \cdots \to A_{in_i} \to T$$

where some of the argument types $A_{i,j}$ may involve T itself, allowing recursion. There are some restrictions on how T can appear in a constructor, but we will postpone this discussion until after some basic examples of inductive types – for many common inductive types, the simplest form of occurrence $A_{i,j} \equiv T$ is already sufficient.

A constructor not dependent on T is called a *base constructor*, since it gives non-recursive ways to form terms of T, acting as a base case for the inductive construction of other terms. Base constructors guarantee that the type is non-empty, acting as leaves in the constructor tree.

Conversely, to eliminate a term of an inductive type, we use the fact that every term is built from a well-founded tree of constructors. This allows us to define functions out of the type by structural recursion; to define a function $T \to B$, it suffices to specify how the function behaves on each constructor. The resulting function, called the *recursor*, then extends to all terms of T by structural recursion, and the computation rules of the inductive type then ensure that the recursor reduces as expected as it breaks down the constructor tree.

- Formation: we will include every inductive type that does not involve other types in its constructors as a base type in our type theory. That is, every inductive type T whose constructors all only involve T satisfies the type formation judgement witout any assumptions:

$$\frac{}{T \ \mathsf{type}}$$

- Introduction: for each constructor c_i, we include an introduction rule of the form

$$\frac{\Gamma \vdash a_1 : A_{i,1} \quad \cdots \quad \Gamma \vdash a_{n_i} : A_{i,n_i}}{\Gamma \vdash c_i(a_1, \ldots, a_{n_i}) : T}$$

  That is, we can construct terms of T by applying the constructors to appropriate arguments, possibly recursively, if some of the $A_{i,j}$ involve T.

- Elimination: since a term of an inductive type is determined by a well-founded tree of constructors, we can define a function $f : T \to B$ by specifying its behaviour on each constructor, including recursively built terms.

  Formally, for each constructor

$$c_i : A_{i,1} \to A_{i,2} \to \cdots \to A_{i,n_i} \to T$$

  we require the provision of a *step function* or *branch*:

$$f_i : (A_{i,1})' \to (A_{i,2})' \to \cdots \to (A_{i,n_i})' \to B$$

  where

$$(A_{i,j})' \coloneqq A_{i,j}[B/T]$$

  replaces any recursive arguments of type T by B, since the recursion will supply B-values for them. In particular,

  - if c_i is a base constructor, the step function f_i doesn't require any recursion, corresponding to the base case;
  - if c_i has recursive arguments, then f_i specifies how to combine the values from recursion on those arguments to compute $f(c_i(\ldots))$.

  The *recursor*

$$\mathsf{rec}_T^B : \big((A_{i,1})' \to \cdots \to (A_{i,n_i})' \to B)\big) \to \cdots \to \big((A_{k,1})' \to \cdots \to (A_{k,n_k})' \to B\big) \to B \to A$$

  is then introduced by:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash f_i : (A_{i,1})' \to \cdots \to (A_{i,n_i}) \text{ for } i = 1, \ldots, k}{\Gamma \vdash \mathsf{rec}_T^B(f_1, \ldots, f_k)(t) : B}$$

  (We will generally not write out the introduction rule for the recursor in the future unless relevant, as it is of the same shape in all cases.)

- Computation: for each constructor c_i, we have:

$$\mathsf{rec}_T^B(f_1, \ldots, f_k)\big(c_i(a_1, \ldots, a_{n_i})\big) \equiv f_i(a_1', \ldots, a_{n_i}')$$

  where a_j' is defined by lifting the recursion along the structure of $A_{i,j}$. Formally, for a fixed type X, we define the map $\mathsf{lift}_X : (T \to B) \to X \to X[B/T]$ by:

$$\mathsf{lift}_X(h)(x) = \begin{cases} x & X \text{ does not involve } T, \\ h(x) & X = T, \\ (\mathsf{lift}_{X_1}(h)(x_1), \mathsf{lift}_{X_2}(h)(x_2)), & X = X_1 \times X_2,\ x = (x_1, x_2), \\ \mathsf{inl}(\mathsf{lift}_{X_1}(h)(y)), & X = X_1 + X_2,\ x = \mathsf{inl}(y), \\ \mathsf{inr}(\mathsf{lift}_{X_2}(h)(y)), & X = X_1 + X_2,\ x = \mathsf{inr}(y), \\ \quad\vdots & \quad\vdots \end{cases}$$

  where lift_X recurses over all of the constructors of X.

Then, we transform inputs to the step functions as:

$$a'_j :\equiv \begin{cases} a_j & A_{i,j} \text{ does not involve } T \\ \mathsf{lift}_{A_{i,j}}\big(\mathsf{rec}^B_T(f_1,\ldots,f_k)\big)(a_j) & \text{otherwise} \end{cases}$$

(The case analysis is unnecessary here, and is included only for clarity; if $A_{i,j}$ does not involve T, then $\mathsf{lift}(\mathsf{rec})$ is just the identity, so both branches agree that $a'_j \equiv a_j$.)

The reason we cannot immediately replace a_j by $\mathsf{rec}(a_j)$ alone is that rec returns a value of the target type B, while $A_{i,j}$ may involve T nested deeper than the top level, within other type formers, so the resulting argument supplied to f_i would not type check.

Instead, we recursively traverse the structure of $A_{i,j}$, applying rec to every subterm of type T to correctly obtain a value of type $A_{i,j}[B/T]$. For a product type $X_1 \times X_2$, lift traverses each branch, applying itself to each component; for sums $X_1 + X_2$, it applies to the contained value, preserving the injection; and so on.

Note also that lift_X pattern matches over *types* and not values, so this construction, as written, is technically metatheoretic for now. However, we only have to include constructors for types that actually occur in a constructor, and since recursive types can only have finitely many constructors, we could manually inline all recursive calls of lift_X at every argument of type T, so this is definable internally to the theory. For simplicity, however, we will retain the use of lift_X as a shorthand for this inlining construction.

Moreover, if we only have occurrences of T at the top level ($A_{i,j} \equiv T$), then just applying rec would be sufficient since lift immediately applies rec in these cases – many of the examples we will look at will only have this simple pattern, in which case will not explicitly invoke the lift function at all.

- Uniqueness: since terms of an inductive type T are generated solely by its constructors, equality of terms is determined by induction on the constructor tree. In particular, two terms of an inductive type T are judgementally equal if and only if they are built from the same constructor applied to judgementally equal arguments.

  That is, if c_i is a constructor of T, then:

$$\frac{\Gamma \vdash c_i(a_1,\ldots,a_{n_i}) \equiv c_i(b_1,\ldots,b_{n_i}) : T}{\Gamma \vdash a_j \equiv b_j : A_{i,j}}$$

  for $j = 1,\ldots,n_i$. Moreover, distinct constructors produce distinct terms:

$$\frac{}{\Gamma \vdash c_i(a_1,\ldots,a_{n_i}) \not\equiv c_j(b_1,\ldots,b_{n_j}) : T}$$

  for $i \neq j$. Note that this holds true even if the constructors take the same arguments.

Note that the uniqueness principles are *emergent* properties that arise from the inductive definition: they are not postulated as separate axioms, but follow automatically from the syntactic structure of constructor trees and the definition of equality on terms of T:

Theorem 2.5 (Constructor Discrimination)**.** *If $i \neq j$, then for all argument tuples $a_1,\ldots,a_{n_i}$ and $b_1,\ldots,b_{n_j}$:*

$$c_i(a_1,\ldots,a_{n_i}) \not\equiv c_j(b_1,\ldots,b_{n_j})$$

Proof. Fix i, and let $\mathsf{Bool} :\equiv \mathsf{unit} + \mathsf{unit}$ be the type with two inhabitants, labelled $\top$ and $\bot$. We define a function $f : T \to \mathsf{Bool}$ with the constant step functions

$$f_i \equiv \lambda a_1 \ldots a_{n_i}.\top$$

and

$$f_j \equiv \lambda b_1 \ldots b_{n_i}.\bot$$

for all $j \neq i$. By the computation rules, we have

$$f\big(c_i(a_1, \ldots, a_{n_i})\big) \equiv \top, \qquad f\big(c_j(b_1, \ldots, b_{n_j})\big) \equiv \bot$$

If we had $c_i(a_1, \ldots, a_{n_i}) \equiv c_j(b_1, \ldots, b_{n_j})$, then applying f to both sides would yield $\top \equiv \bot$, which is a contradiction, as $\top$ and $\bot$ are distinct terms of Bool. ■

The existence of the eliminator permits us to distinguish constructor cases by producing different values in an appropriate discriminating target type; the computation rule makes those distinctions definitional on canonical forms, so equalities across different constructor forms are impossible.

To prove injectivity, we

Lemma 2.6. *The sum constructors are injective.*

Theorem 2.7 (Injectivity of Constructors)**.** *If c_i is a constructor of T, then:*

$$\frac{\Gamma \vdash c_i(a_1, \ldots, a_{n_i}) \equiv c_i(b_1, \ldots, b_{n_i}) : T}{\Gamma \vdash a_j \equiv b_j : A_{i,j}}$$

for $j = 1, \ldots, n_i$.

Proof. Let

$$B :\equiv \big((A_{1,1})' \times \cdots \times (A_{1,n_1})'\big) + \cdots + \big((A_{k,1})' \times \cdots \times (A_{k,n_k})'\big)$$

The idea is that recursion into this type allows us to record the top-level constructor (given by the sum tag in_i) and the tuple of recursively-lifted arguments (given by the products) at each step, from which we can recover componentwise equalities by structural induction.

For each constructor c_i, define the step function to be the ith sum injection:

$$f_i(x'_1, \ldots, x'_{n_i}) :\equiv \mathsf{in}_i\big(\langle x'_1, \ldots, x'_{n_i}\rangle\big)$$

Then, by the computation rule, we have:

$$\mathsf{rec}_T^B(f_1, \ldots, f_i)\big(c_i(a_1, \ldots, a_{n_i})\big) \equiv \mathsf{in}_i\big(\langle a'_1, \ldots, a'_{n_i}\rangle\big)$$

By assumption,

$$c_i(a_1, \ldots, a_{n_i}) \equiv c_i(b_1, \ldots, b_{n_i})$$

By applying the recursor to both sides and using congruence of equality under function application, we obtain:

$$\mathsf{in}_i\big(\langle a'_1, \ldots, a'_{n_i}\rangle\big) \equiv \mathsf{in}_i\big(\langle b'_1, \ldots, b'_{n_i}\rangle\big)$$

and by injectivity of sum injections and componentwise equality, we have $a'_j \equiv b'_j$ for all $1 \leq j \leq n_i$. We now show that equality descends through the lifting operation $(-)' \equiv \mathsf{lift}(\mathsf{rec})$ by structural induction on the height of $A_{i,j}$ to deduce that $a_j \equiv b_j$.

- Base case 1: $A_{i,j}$ does not involve T. The lift is then the identity, so we have $a_j \equiv b_j$.
- Base case 2: $A_{i,j} = T$. Then, $A_{i,j}[B/T] \equiv B$,

  $$a'_j \equiv \mathsf{rec}_T^B(f_1, \ldots, f_k)(a_j)$$
  $$b'_j \equiv \mathsf{rec}_T^B(f_1, \ldots, f_k)(b_j)$$

  By the computation rule of rec, and the choice of B and f_i, rec sends each term $c_i(\vec{t})$ to $\mathsf{in}_i(\langle t'_1, \ldots, t'_{n_i}\rangle)$ (i.e. the ith summand carrying the lifted tuple of arguments). Because the summands of B are disjoint, $a'_j \equiv b'_j$ implies

  – a_j and b_j have the same top-level constructor c_k;

  $$a_j \equiv c_k(a_{j,1}, \ldots, a_{j,n_k}), \qquad b_j \equiv c_k(b_{j,1}, \ldots, b_{j,n_k})$$

– and moreover, the corresponding tuples of *lifted* subarguments are equal:

$$\langle a'_{j,1},\ldots,a'_{j,n_k}\rangle \equiv \langle b'_{j,1},\ldots,b'_{j,n_k}\rangle$$

Comparing components, we have $a'_{j,\ell} \equiv b'_{j,\ell}$ for each ℓ. Each of these subarguments are of type $A_{k,\ell}[B/T]$, where the types $A_{k,\ell}$ are strictly structurally-smaller occurrences than the whole term c_k, so by the induction hypothesis, equality descends to $a_{j,\ell} \equiv b_{j,\ell}$ for every ℓ. Finally, by congruence of equality under constructor application,

$$a_j \equiv c_k(a_{j,1},\ldots,a_{j,n_k}) \equiv c_k(b_{j,1},\ldots,b_{j,n_k}) \equiv b_j$$

- Products: $A_{i,j} \equiv X_1 \times X_2$. Then, $A_{i,j}[B/T] = X_1[B/T] \times X_2[B/T]$, with

$$a'_j \equiv \langle \mathsf{lift}_{X_1}(\mathsf{rec})(a_{j,1}),\mathsf{lift}_{X_2}(\mathsf{rec})(a_{j,2})\rangle$$
$$b'_j \equiv \langle \mathsf{lift}_{X_1}(\mathsf{rec})(a_{j,1}),\mathsf{lift}_{X_2}(\mathsf{rec})(a_{j,2})\rangle$$

From $a'_j \equiv b'_j$, we have the component equalities:

$$\mathsf{lift}_{X_1}(h)(a_{j,1}) \equiv \mathsf{lift}_{X_1}(h)(b_{j,1})$$
$$\mathsf{lift}_{X_2}(h)(a_{j,2}) \equiv \mathsf{lift}_{X_2}(h)(b_{j,2})$$

so by the inductive hypothesis, $a_{j,1} \equiv b_{j,1}$ and $a_{j,2} \equiv b_{j,2}$, and hence $a_j \equiv b_j$.

- Sums: $A_{i,j} \equiv X_1 + X_2$. Then, $A_{i,j}[B/T] \equiv X_1[B/T] + X_2[B/T]$. Suppose $a'_j \equiv \mathsf{inl}(\mathsf{lift}_{X_1}(\mathsf{rec})(x))$ and $b'_j \equiv \mathsf{inl}(\mathsf{lift}_{X_1}(\mathsf{rec})(y))$. By assumption, $a'_j \equiv b'_j$, so

$$\mathsf{lift}_{X_1}(\mathsf{rec})(x) \equiv \mathsf{lift}_{X_1}(\mathsf{rec})(y)$$

and by the inductive hypothesis, $x \equiv y$, giving $a_j \equiv b_j$. The proof for inr is identical.

- The argument continues similarly for any other constructor of T: equality of lifted terms reduces to equalities of smaller subterms, and the inductive hypothesis on type structure yields equality of the originals.

The structural induction shows that for every $A_{i,j}$, the map $\mathsf{lift}_{A_{i,j}}(\mathsf{rec})$ is injective, and hence we obtain $a_j \equiv b_j$ for $j = 1,\ldots,n_i$. ■

These two results together are called the *inversion lemma for constructors* or the *no-confusion principle for inductive types.*

Proof. Fix an index i and position $j \leq n_i$, and let $B :\equiv A_{i,j}$. If B is empty, the result is vacuous. Otherwise, let $\bullet : B$.

For the distinguished constructor c_i, define the step function

$$f_\ell(a_1, \ldots, a_{n_i}) \equiv \begin{cases} \mathsf{lift}_{A_{i,j}}\big(\mathsf{rec}_T^B(f_1, \ldots, f_{n_i})\big)(a_j) & \ell = i \\ \bullet & \text{otherwise} \end{cases}$$

By the computation rule,

$$\mathsf{rec}_T^B(f_1, \ldots, f_k)\big(c_i(a_1, \ldots, a_{n_i})\big) \equiv f_i(a'_1, \ldots, a'_{n_i}) \equiv \mathsf{lift}_{A_{i,j}}\big(\mathsf{rec}_T^B(f_1, \ldots, f_k)\big)(a_j)$$

Similarly,

$$\mathsf{rec}_T^B(f_1, \ldots, f_k)\big(c_i(b_1, \ldots, b_{n_i})\big) \equiv f_i(b'_1, \ldots, b'_{n_i}) \equiv \mathsf{lift}_{A_{i,j}}\big(\mathsf{rec}_T^B(f_1, \ldots, f_k)\big)(b_j)$$

By assumption, we have

$$c_i(a_1, \ldots, a_{n_i}) \equiv c_i(b_1, \ldots, b_{n_i})$$

so by applying the recursor to both sides:

$$\mathsf{lift}_{A_{i,j}}\big(\mathsf{rec}_T^B(f_1, \ldots, f_k)\big)(a_j) \equiv \mathsf{lift}_{A_{i,j}}\big(\mathsf{rec}_T^B(f_1, \ldots, f_k)\big)(b_j)$$

We now show that $\mathsf{lift}(\mathsf{rec})$ is injective by structural induction on $A_{i,j}$.

- Base case 1: $A_{i,j}$ does not involve T. This lift is then the identity, so $a_j \equiv b_j$ immediately.
- Base case 2: $A_{i,j} = T$. Then, $A_{i,j}[B/T] \equiv B$,
- Products: $A_{i,j} \equiv X_1 \times X_2$, with $a_j \equiv \langle a_{j,1}, a_{j,2}\rangle$ and $b_j \equiv \langle b_{j,1}, b_{j,2}\rangle$. Then,

  $$\mathsf{lift}_{X_1}(h)(a_{j,1}) \equiv \mathsf{lift}_{X_1}(h)(b_{j,1})$$
  $$\mathsf{lift}_{X_1}(h)(a_{j,2}) \equiv \mathsf{lift}_{X_1}(h)(b_{j,2})$$

  so by the inductive hypothesis, $a_{j,1} \equiv b_{j,1}$ and $a_{j,2} \equiv b_{j,2}$, and hence $a_j \equiv b_j$.
- Sums: $A_{i,j} \equiv X_1 + X_2$. Suppose $a_j \equiv \mathsf{inl}(x)$ and $\mathsf{inl}(y)$ (the proof for inr is identical). Then,

  $$\mathsf{lift}_{X_1}(h)(x) \equiv \mathsf{lift}_{X_1}(h)(y)$$

  by function extensionality of inl, so $x \equiv y$ by the induction hypothesis, and hence $a_j \equiv b_j$.
- Repeat this argument for every other constructor of T.

■

Example. The nat type can be constructed as a simple inductive type, with two constructors of arity 0 and 1:

$$0 : \mathsf{nat} \qquad \text{and} \qquad \mathsf{succ} : \mathsf{nat} \to \mathsf{nat}$$

Following the recipe above, we require step functions $b :\equiv f_1 : B$ and $r :\equiv f_2 : B \to B$, with the recursor given by:

$$\frac{\Gamma \vdash n : \mathsf{nat} \qquad \Gamma \vdash b : B \qquad \Gamma \vdash r : B \to B}{\Gamma \vdash \mathsf{rec}_{\mathsf{nat}}^B(b, r)(t) : B}$$

Then, the computation rules for $\mathsf{natrec} :\equiv \mathsf{rec}_{\mathsf{nat}}^B(b, r)$ are:

$$\mathsf{natrec}(0) \equiv b_0$$
$$\mathsf{natrec}(\mathsf{succ}(n)) \equiv r(\mathsf{natrec}(n))$$

which is precisely the definition of simple recursion. △

In traditional first-order formulations of Peano arithmetic, we also begin with two "constructors", $0 \in \mathbb{N}$, and $n \in \mathbb{N} \Rightarrow \mathsf{succ}(n) \in \mathbb{N}$. However, we must additionally axiomatically assert:

- the injectivity of constructors:

$$\mathsf{succ}(n) = \mathsf{succ}(m) \implies n = m$$

- and the disjointness of constructors:

$$\mathsf{succ}(n) \neq 0$$

However, for the type nat, these properties follow *definitionally* from the way inductive types are defined. Since an inductive type T is defined freely from its constructors, the only terms of type T are those are those obtained by finitely applying these constructors, and hence two terms of T are judgementally equal if and only if they are syntactically identical, up to the computation rules and any other applicable judgemental equalities.

For nat, we have as *judgemental properties* of the syntax,

$$0 \not\equiv \mathsf{succ}(n) \qquad \text{and} \qquad \mathsf{succ}(n) \equiv \mathsf{succ}(m) \iff n \equiv m$$

This example captures the essence of what it means for a type to be inductive: it is freely generated by a finite collection of constructors, and all of its properties follow from the syntactic structure these constructors impose; consequently, equalities within the type are entirely determined by the structure of constructor expressions, rather than by any separate axioms.

It is instructive to observe that this notion of freeness applies even to the simplest possible types. The pattern of constructors, recursion, and computation that we have seen for nat also appears, in degenerate form, in the case of types with no recursive arguments. For instance, unit and void.

The unit type is a trivial inductive type, with a single constructor of arity 0:

$$\star : \mathsf{unit}$$

Following the general pattern, the recursor $\mathsf{elim} :\equiv \mathsf{rec}^B_{\mathsf{unit}} : B \to \mathsf{unit} \to B$ for unit is defined for any type B by specifying a single step value $b : B$:

$$\frac{\Gamma \vdash t : \mathsf{unit} \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{elim}(b,t) : B}$$

The computation rule is then what we had before:

$$\mathsf{elim}(b,\star) \equiv b$$

Since $*$ is the only constructor, we also obtain the uniqueness property:

$$t \equiv \star$$

for any $t : \mathsf{unit}$, as before.

The void type is another trivial inductive type, with no constructors. The recursor $\mathsf{abort} = \mathsf{rec}^B_{\mathsf{void}} : \mathsf{void} \to B$ is defined by giving step functions for each constructor, but there are no constructors, so there are no such functions to specify, and hence the elimination rule for void has no other premises, reading as:

$$\frac{\Gamma \vdash t : \mathsf{void}}{\Gamma \vdash \mathsf{rec}^B_{\mathsf{void}}(t) : B}$$

There are no computation rules to specify, because there are no terms of type void.

Here is an example of a new inductive type that we have not seen before. Given a type A, let $\mathsf{List}(A)$ be the inductive type with constructors:

$$\mathsf{nil}_A : \mathsf{List}(A) \qquad \text{and} \qquad \mathsf{cons}_A : A \to \mathsf{List}(A) \to \mathsf{List}(A)$$

Intuitively, a list is either the empty list, nil_A, or is formed by adjoining a term to the front of another list, $\mathsf{cons}_A(a,\ell)$, where $a : A$ and $\ell : \mathsf{List}(A)$. Note that, unlike the other inductive types we have seen so far, this type is *parametrised* by A:

$$\frac{A\ \mathsf{type}}{\mathsf{List}(A)\ \mathsf{type}}$$

For every type A, we can form a corresponding list type.

To construct a function $\mathsf{List}(A) \to B$ out of a list, we require step functions $b : B$ and $f : A \to B \to B$. The recursor then satisfies the following computation rules:

$$\mathsf{rec}^B_{\mathsf{List}(A)}(b,f)(\mathsf{nil}_A) \equiv b : B$$
$$\mathsf{rec}^B_{\mathsf{List}(A)}(b,f)\big(\mathsf{cons}_A(a,\ell)\big) \equiv f\big(a,\mathsf{rec}^B_{\mathsf{List}(A)}(b,f)(\ell)\big) : B$$

For example, we can define the function $\mathsf{len} : \mathsf{List}(A) \to \mathsf{nat}$ that returns the length of a list by providing the step functions

$$f_1 :\equiv 0 : \mathsf{nat}$$
$$f_2 :\equiv \lambda a.\lambda n.\mathsf{succ}(\mathsf{n}) : A \to \mathsf{nat} \to \mathsf{nat}$$

Then, $\mathsf{len} :\equiv \mathsf{rec}^{\mathsf{nat}}_{\mathsf{List}(A)}(f_1,f_2) : \mathsf{List}(A) \to \mathsf{nat}$ satisfies

$$\mathsf{len}(\mathsf{nil}_A) \equiv 0 : \mathsf{nat}$$
$$\mathsf{len}\big(\mathsf{cons}_A(a,\ell)\big) \equiv \mathsf{succ}\big(\mathsf{len}(\ell)\big) : \mathsf{nat}$$

We can also define the function $\mathsf{map} : (A \to B) \to \mathsf{List}(A) \to \mathsf{List}(B)$ that applies a function $g : A \to B$ pointwise to a list of type A with these step functions:

$$f_1 :\equiv \mathsf{nil}_B : \mathsf{List}(B)$$
$$f_2 :\equiv \lambda a.\lambda \ell'.\mathsf{cons}_B\big(g(a),\ell'\big) : A \to \mathsf{List}(B) \to \mathsf{List}(B)$$

We note that the step function f_2 depends on the function $g : A \to B$. To define the general map function that takes g as argument, note that our context currently includes g, and f_2 is being defined in that extended context:

$$g : A \to B \vdash f_2 : A \to \mathsf{List}(B) \to \mathsf{List}(B)$$

so we can abstract over g to obtain the general map function:

$$\mathsf{map} :\equiv \lambda g : A \to B.\mathsf{rec}^{\mathsf{List}(B)}_{\mathsf{List}(A)}(f_1,f_2) : (A \to B) \to \mathsf{List}(A) \to \mathsf{List}(B).$$

This definition satisfies the computation rules:

$$\mathsf{map}(g,\mathsf{nil}_A) \equiv \mathsf{nil}_B : \mathsf{List}(B)$$
$$\mathsf{map}(g,\mathsf{cons}_A(a,\ell)) \equiv \mathsf{cons}_B\big(g(a),\mathsf{map}(g,\ell)\big).$$

That is, mapping g over the empty A-list returns the empty B-list; and mapping g over the A-list with head a and tail ℓ returns the B-list with head $g(a)$ and tail given by recursively mapping g over ℓ.

In fact, every type former we have seen so far, apart from function types, can be defined as an inductive type.

Example. The product type $A \times B$ is an inductive type, with one constructor:

$$\mathsf{pair} : A \to B \to A \times B$$

and recursor satisfying:

$$\mathsf{rec}^{C}_{A\times B}(f)\big(\mathsf{pair}(a,b)\big) \equiv f(a,b)$$

for step function $f : A \to B \to C$.

Our previous definition of the product is equivalent to this inductive definition, in the sense that fst and snd can be recovered from this inductive type's recursor,

$$\mathsf{fst} :\equiv \lambda p.\mathsf{rec}^{A}_{A\times B}(\lambda a.\lambda b.a)(p)$$
$$\mathsf{snd} :\equiv \lambda p.\mathsf{rec}^{A}_{A\times B}(\lambda a.\lambda b.a)(p)$$

and conversely, the recursor can be recovered from fst and snd:

$$\mathsf{rec}^{C}_{A\times B} :\equiv \lambda f.\lambda p.f(\mathsf{fst}(p),\mathsf{snd}(p))$$

△

Example. Sum types can be defined inductively via the constructors:

$$\mathsf{inl} : A \to A + B \qquad \text{and} \qquad \mathsf{inr} : B \to A + B$$

with recursor satisfying

$$\mathsf{rec}^{C}_{A+B}(f_1,f_2)(\mathsf{inl}(a)) \equiv f_1(a)$$
$$\mathsf{rec}^{C}_{A+B}(f_1,f_2)(\mathsf{inr}(b)) \equiv f_1(b)$$

for step functions $f_1 : A \to C$ and $f_2 : B \to C$. △

2.4.1 Polarity

Why is the function type not an inductive type? We have previously mentioned that not every occurrence of a type in a constructor is permissible. This is because certain patterns of self-reference can lead to non-terminating or inconsistent definitions.

For instance, suppose we attempted to define an "inductive" type T with the constructors

$$t : T \qquad \text{and} \qquad \mathsf{foo} : (T \to T) \to T$$

Consider the function $h : T \to T$ defined by

$$h(x) :\equiv \mathsf{foo}(\lambda y.x)$$

Now, define

$$\Omega :\equiv \mathsf{foo}(h)$$

Expanding h, we have

$$\begin{aligned}\Omega &\equiv \mathsf{foo}(\lambda x.\mathsf{foo}(\lambda y.x))\\ &\equiv \mathsf{foo}(\lambda x.\Omega)\end{aligned}$$

$$f : T \to T :\equiv \mathsf{foo}(\lambda x.\Omega)$$

2.5 Derivations

2.6 The Curry-Howard Isomorphism

For readers familiar with sequent calculus

The similarity with first-order logic sequents $\Gamma \vdash \varphi$ – "φ is provable under assumptions Γ" – is not a coincidence. Just as logical inference rules allow us to derive new formulae from existing assumptions, *typing inference rules* allows us to derive new typing judgements from existing ones.

In fact, simple type theory only has one primitive inference rule:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}\,(\text{axiom})$$

That is, if we assume x has type A, then we can derive that x has type A.

Theorem 2.8. *The weakening rule is admissible*

$$\frac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}\,(\text{wk})$$

Proof. By induction on the derivation of $\Gamma \vdash x : A$. ■

Theorem 2.9. *The exchange rule rule is admissible*

$$\frac{\Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, x : A, \Delta \vdash t : C}\,(\text{xch})$$

Consider all the introduction and elimination rules we have seen so far:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,(\to_{\text{I}}) \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f(x) : B}\,(\to_{\text{E}})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b\rangle : A \times B}\,(\times_{\text{I}}) \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{fst}(p) : A}\,(\times_{\text{E}_1}) \quad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{snd}(p) : B}\,(\times_{\text{E}_2})$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}(a) : A + B}\,(+_{\text{I}_1}) \quad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}(a) : A + B}\,(+_{\text{I}_2}) \qquad \frac{\Gamma \vdash t : A + B \qquad \Gamma \vdash f : A \to C \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathsf{case}(t; f, g) : C}\,(+_{\text{E}})$$

$$\frac{}{\cdot \vdash \star : \mathsf{unit}}\,(\mathsf{unit}_{\text{I}}) \qquad \frac{\Gamma \vdash x : \mathsf{void}}{\Gamma \vdash \mathsf{abort}(x) : A}\,(\mathsf{void}_{\text{E}})$$

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}\,(\text{axiom})$$

$$\frac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}\,(\text{weakening})$$

$$\frac{\Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, x : A, \Delta \vdash t : C}\,(\text{exchange})$$

$$\frac{\Gamma, x : A, y : A, \Delta \vdash t : B}{\Gamma, x : A, \Delta \vdash t[x/y] : B}\text{(substitution)}$$

$$\frac{\Gamma, x : A, x : A, \Delta \vdash y : B}{\Gamma, x : A, \Delta \vdash y : B}\text{(contraction)}$$

$$\frac{\Gamma \vdash x : A \qquad \Delta, x : A \vdash y : B}{\Gamma, \Delta \vdash y : B}\text{(cut)}$$

Theorem 2.10 (Curry-Howard Correspondence)**.** *Given a context* Γ *and a type* A*, term erasure induces an correspondence between terms of type* A *in context* Γ*, and intuitionistic proofs of* $\Gamma \vdash A$*.*

Proof. Suppose we have a proof π of a sequent $\Gamma \vdash A$. ■

In addition to term erasure, one can consider type erasure, where the type annotations are removed from terms, leaving only the underlying computational structure. Type erasure illustrates that the dynamics of computation in the simply typed λ-calculus is independent of types: types guide construction and correctness, but do not affect the actual reduction of terms. While term erasure reveals the Curry-Howard correspondence, type erasure connects the theory to operational behaviour in programming languages.

At this point it is worth stressing the difference between the set-theoretic and the type-theoretic treatments. In set theory, the Kuratowski construction forces us to prove a separate "extensionality lemma" for ordered pairs: if $\langle a,b\rangle = \langle c,d\rangle$ as sets, then necessarily $a = c$ and $b = d$. In *simple type theory*, however, pairs are taken as primitive terms of the product type. The projections are governed by definitional equalities

$$\pi_1\langle a,b\rangle \equiv a, \qquad \pi_2\langle a,b\rangle \equiv b,$$

and the equality

$$\langle a,b\rangle = \langle c,d\rangle$$

is itself definitionally equivalent to the conjunction $a = c$ and $b = d$. No additional extensionality proof is required: the injectivity of pairing is already built into the definitional equality rules of the type system.

In this paper, we will give an introduction to type theory as a foundation for mathematics. Our goal is not to argue against set theory, but rather to show how type theory provides a different perspective: one where the "grammar" of mathematics is enforced by the system itself, and where the boundaries between logic and mathematics begin to dissolve. We will focus in particular on dependent and inductive types, which together provide a flexible and expressive framework suitable for most ordinary mathematical practice.

To make this distinction concrete, consider the natural numbers.

Numbers in ZFC (with extrinsic typing). In ZFC set theory, the natural numbers are defined *indirectly* as a particular set. For instance, we can take $\mathbb{N}$ to be the least inductive set: the intersection of all sets containing $\varnothing$ and closed under the successor operation $x \mapsto x \cup \{x\}$. In this presentation, the number 0 is identified with the empty set $\varnothing$, the number 1 with $\{0\}$, the number 2 with $\{0,1\}$, and so on. Thus, each numeral is a complicated set in disguise. If we then want to use these as "numbers," we add an external convention: we agree that the intended type of these sets is $\mathbb{N}$, and we restrict ourselves to asking number-theoretic questions about them. The typing discipline here is *extrinsic*: the set $\{0,1\}$ exists regardless of whether we choose to regard it as the number 2, a two-element set, or something else entirely.

Numbers in type theory (with intrinsic typing). In type theory, by contrast, the natural numbers are introduced directly as a new *type*, with their behaviour given by constructors:

$$0 : \mathbb{N}, \qquad \mathsf{succ} : \mathbb{N} \to \mathbb{N}.$$

That is, 0 is a natural number, and if n is a natural number, then so is $\mathsf{succ}(n)$. The only inhabitants of $\mathbb{N}$ are those built by finitely many applications of these constructors. In this way, there is no need to encode numbers as sets, nor to impose a typing discipline externally: the typing is *intrinsic*. An object is a number if and only if it is of type $\mathbb{N}$.

Comparison. In set theory, then, $\mathbb{N}$ is a particular subset of the universe of all sets, and $n \in \mathbb{N}$ is a *predicate* that tells us whether a given set encodes a number. In type theory, $\mathbb{N}$ is not a set but a *type*, and the statement $n : \mathbb{N}$ is not a predicate but a *judgment* telling us that n is a number. The difference is the same as between "here is an untyped object which happens to satisfy the condition of being a number" (extrinsic typing) and "here is an intrinsically typed object, a number from the start."

This example illustrates how type theory enforces the grammar of mathematics intrinsically, while set theory relies on extrinsic conventions layered on top of a uniform background of sets.

3 The Curry-Howard Isomorphism

3.1 Sequent Calculus

3.2 Type Erasure

4 Dependent Type Theory

$$\Gamma \vdash A \ \mathsf{type}$$

Since we now treat terms and types more similarly, it will streamline our theory to introduce a *type of types*, or a *universe type*, Type (sometimes also written as $\mathcal{U}$, to match with similar universes in set theory and logic), in which case the type judgement

$$\Gamma \vdash A \ \mathsf{type}$$

takes the same shape as an ordinary typing judgement $\Gamma \vdash a : A$;

$$\Gamma \vdash A : \mathsf{Type}$$

However, this only raises the question, what is the type of Type? Since it is the type of types, it is itself a type, so perhaps we have the following:

$$\mathsf{Type} : \mathsf{Type}$$

However, this choice of type will allow us to quantify over all types with dependent types, and in doing so, a type-theoretic analogue of the Bureli-Forti paradox occurs if we allow this, as we will show later.

Rather than adding a single universe type, we will add an infinite hierarchy of universe types, each being an inhabitant of the next:

$$\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \mathsf{Type}_3 : \cdots$$

For our purposes, we will rarely need to work with any higher universe levels, so we will abbreviate Type_0 to just Type when the context is clear.

4.1 Dependent Sums

4.2 Dependent Products

4.3 Inductive Types

An *inductive type* is, roughly speaking, a way of introducing base types in terms of constants and functions that create terms of that type. For instance, the natural numbers are a basic inductive type, with two constructors

$$\frac{}{\Gamma \vdash 0 : \mathsf{nat}} \qquad \frac{\Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(n) : \mathsf{nat}}$$

4.4 Quantifiers

4.5 Girard's Paradox

Addendum

5 Metatheory

This section develops certain metatheoretic properties of the reduction and equality relations. The results here will be used later to justify properties such as confluence, canonical forms, and the inversion lemma, but they are not required to follow the basic definitions and computational rules of the type theory.

It is strongly recommended that a new reader primarily interested in understanding the operational behaviour of the type theory should skip this section on first reading. All subsequent sections can be followed without reference to these metatheoretic details; the results presented here serve mainly to justify later formal arguments.

Also, for the purposes of this metatheoretic development only, we shall consider the existence of terms independently of their types. That is, *for this section only*, we work extrinsically: terms are formed according to the raw syntax of the calculus, and typing is considered as a separate judgement relating terms to types.

This approach departs from the intrinsic typing discipline advocated for elsewhere in this paper, where only well-typed terms are considered meaningful. The extrinsic perspective is convenient here because standard arguments for strong normalisation, confluence, and canonical forms only rely on reasoning about reduction sequences and normal forms of arbitrary syntactic terms; such reasoning does not require that every term be well-typed from the outset. Once these metatheoretic results are established, they will be applied to well-typed terms in the usual, intrinsically typed setting.

5.1 Operational Semantics Judgemental Equality

In the preceding sections, we introduced the basic type formers of our theory – products, sums, and function types – together with their associated term formation, introduction, elimination, and computation rules. These rules jointly determine the notion of definitional equality $\equiv$, which expresses when two terms are equal in terms of the computational content of the theory. As emphasised above, this equality is fixed syntactically by the canonical ways in which terms of each type may be formed and reduced.

In an earlier section, we treated definitional equality $\equiv$ as a primitive judgement, equipped with the standard inference rules for reflexivity, symmetry, and transitivity, together with stability under substitution and α-conversion.

From the metatheoretic perspective, this judgement can also be understood *operationally* in terms of the reduction relation underlying each type former. Specifically, we may define $a \equiv b$ to hold if and only if a and b both reduce to a common term c.

In a well-behaved type theory – one in which reduction is *terminating* and *confluent* – these two presentations coincide: every equality derivable from the inference rules is witnessed by reduction to a common normal form, and conversely, every pair of terms with a common normal form is derivably equal. This identification allows us to reason about definitional equality both syntactically, via the primitive inference rules, and computationally, via reduction and canonical forms.

Although judgemental equality is defined by syntactic rules, we have thus far relied only informally on its basic properties. For example, in the case of product types, we argued that two pairs are equal if and only if their components are equal. A fully rigorous justification of such statements requires certain metatheoretic facts about the reduction relation that underlies definitional equality, in particular:

- that reduction is *terminating*, so that every term reduces to a *normal form*;
- that reduction is *confluent*, so that every reduction path reaches the same unique normal form;

- and that normal forms exhibit a "canonical shapes" property, determined by their type.

These properties together entail that definitional equality admits a normal form characterisation: two terms are definitionally equal if and only if they reduce to the same canonical form.

Before giving the formal definition of reduction, it is useful to recall its conceptual role. The definitional equality judgement $t \equiv u$ is intended to identify terms that are equal by computation. Computation in type theory is governed by the elimination rules: when a constructor meets its corresponding eliminator, the resulting redex simplifies. For example, applying a function to an argument computes by β-reduction, and projecting a pair computes by projection rules. Reduction therefore expresses the elementary computational steps available in the theory, while definitional equality is the closure of these steps under symmetry, reflexivity, transitivity, and congruence. This idea is formalised as follows.

The *one-step reduction* relation $\longrightarrow$ is the smallest binary relation on terms closed under all of the β rules:

$$
\begin{aligned}
(\lambda x.t)u &\longrightarrow t[u/x] && (\beta\text{-reduction}) \\
\pi_1\langle a,b\rangle &\longrightarrow a && (\times_{\beta_1}) \\
\pi_2\langle a,b\rangle &\longrightarrow b && (\times_{\beta_2}) \\
\mathsf{case}(\mathsf{inl}(t),\lambda a.f,\lambda b.g) &\longrightarrow f[t/x] && (+_{\beta_1}) \\
\mathsf{case}(\mathsf{inr}(t),\lambda a.f,\lambda b.g) &\longrightarrow g[t/y] && (+_{\beta_2})
\end{aligned}
$$

and is closed under substitution:

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']}$$

where $C[t]$ denotes a term depending on t. That is, the relation is compatible with reductions performed *inside* terms, as well as at the top level. For instance,

$$\frac{t \longrightarrow t'}{\langle s,t\rangle \longrightarrow \langle s,t'\rangle}$$

A term t is then in *normal form* if no β rule applies to it. That is, there is no term t' such that $t \longrightarrow t'$.

Example. Terms of opaque base types are always in normal form, because their types provide no β rules. △

The multi-step reduction relation $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$. That is, the smallest relation such that:

- if $t \longrightarrow t'$, then $t \longrightarrow^* t'$;
- for all terms t, $t \longrightarrow^* t$;
- and if $t \longrightarrow^* t'$ and $t' \longrightarrow^* t''$, then $t \longrightarrow^* t''$.

5.1.1 Normalisation

A *closed* term t is *strongly normalising* it it does not admit an admit reduction sequence

$$t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \cdots$$

Note that if t is strongly normalising, then any t' satisfying $t \to t'$ is also strongly normalising.

Note that this definition could also apply to open terms as a purely syntactic property; however, in metatheoretic results, we usually require a stronger, substitution-closed notion of strong normalisation for open terms.

A term t in context Γ is *strongly normalising* if for every substitution σ mapping each variable $x : A \in \Gamma$ to a strongly normalising closed term $\sigma(x)$, the instantiated term $t[\sigma]$ is strongly normalising.

n the following, we write $\mathsf{SN}(t)$ to denote that t is strongly normalising (syntactically), and for a term t in context Γ, we write $\mathsf{SN}_\Gamma(t)$ to denote that t is strongly normalising under all strongly normalising substitutions for Γ.

Theorem 5.1. *Every well-typed term is strongly normalising. That is, if $\Gamma \vdash t : A$, then $\mathsf{SN}_\Gamma(t)$.*

The proof strategy is standard, following the *reducibility method*. Before we begin, we provide some supplementary definitions and results.

A closed term is *neutral* if it is not a canonical introduction form. That is, it is not an abstraction $\lambda x.t$, a pair $\langle a,b\rangle$, nor a sum injection $\mathsf{inl}(a)/\mathsf{inr}(b)$ for sums. Inductively, this means that a neutral terms an application fn of a function to a neutral term n; a projection $\mathsf{fst}(n)$ of a neutral term n, a case analysis on a neutral term, etc.

For each type A, we define the set $[\![A]\!]$ of *reducibility candidates* of A, such that:

CR1 (Normalisation) Every $t \in [\![A]\!]$ is strongly normalising;

CR2 (Stability under reduction) If $t \in [\![A]\!]$ and $t \to t'$, then $t' \in [\![A]\!]$;

CR3 (Neutral closure) If t is neutral and strongly normalising, and every one-step reduct t' of t satisfies $t' \in [\![A]\!]$, then $t \in [\![A]\!]$.

These sets are defined via structural recursion on A as follows:

- Base types A:
$$[\![A]\!] \coloneqq \{t : A \mid \mathsf{SN}(t)\}$$
- Function types $A \to B$:
$$[\![A \to B]\!] \coloneqq \{f : A \to B \mid \mathsf{SN}(f) \land \forall a \in [\![A]\!].fa \in [\![B]\!]\}$$
so that t behaves well on all reducible arguments:
- Product types $A \times B$:
$$[\![A \times B]\!] \coloneqq \{t : A \times B \mid \mathsf{fst}(t) \in [\![A]\!] \land \mathsf{snd}(t) \in [\![B]\!]\}$$
so that components are reducible;
- Sum types $A + B$:
$$[\![A + B]\!] \coloneqq \left\{ t : A + B \;\middle|\; \begin{array}{r} \mathsf{SN}(t) \land \text{whenever } t \longrightarrow^* \mathsf{inl}(a), a \in [\![A]\!] \\ \land \text{ whenever } t \longrightarrow^* \mathsf{inr}(b), b \in [\![B]\!] \end{array} \right\}$$
so that every reachable injection carries a reducible component.

Because $\longrightarrow$ is closed under contexts, these clauses are sufficient to establish that the three required properties hold by a straightforward induction on types.

Lemma 5.2. *For every type A, the set $[\![A]\!]$ satisfies* CR1, CR2, *and* CR3.

Proof. By induction on A. ∎

Lemma 5.3 (Soundness of Reducibility)**.** *Let $\Gamma \vdash t : A$, and suppose σ is any substitution assigning to each $x : B$ declared in Γ a closed term $\sigma(x) \in [\![B]\!]$. Then, the term $\sigma(t)$ obtained by applying σ to all the free variables of t belongs to $[\![A]\!]$.*

Proof. By induction on the typing derivation of t:

- Variable: if t is x, with $x : B$ declared in Γ, then $\sigma(x) \in [\![B]\!]$ by assumption.

- Abstraction: $t = \lambda x.u$, with $\Gamma, x : A \vdash u : B$. We must show $\sigma(\lambda x.u) = \lambda x.\sigma(u) \in [\![A \to B]\!]$. By the definition of $[\![A \to B]\!]$ we must show: for every $v \in [\![A]\!]$, $(\lambda x.\sigma(u)), v \in [\![B]\!]$. But $(\lambda x.\sigma(u)), v \longrightarrow \sigma(u)[v/x]$ by applying one step of β-reduction. By induction hypothesis applied to u with the substitution σ extended by $x \mapsto v$, we obtain $\sigma(u)[v/x] \in [\![B]\!]$. Because $[\![B]\!]$ is closed under reduction (CR2) we have $(\lambda x.\sigma(u)), v \in [\![B]\!]$, and hence $\lambda x.\sigma(u) \in [\![A \to B]\!]$.
- Application: $t = s\ r$, with $\Gamma \vdash s : A \to B$ and $\Gamma \vdash r : A$. By the induction hypothesis, $\sigma(s) \in [\![A \to B]\!]$, and $\sigma(r) \in [\![A]\!]$. By definition of $[\![A \to B]\!]$, $\sigma(s)\ \sigma(r) \in [\![B]\!]$. That is, $\sigma(t) \in [\![B]\!]$.
- The other constructors and destructors follow the same general pattern:
  - Apply the inductive hypothesis on all immediate subterms to obtain their membership in the corresponding reducibility candidates;
  - Verify that the one-step reducts produced by the term (either by β-rules or by reductions in the subterms) lie in the correct reducibility candidates, using the closure properties.

■

Theorem 5.4 (Strong Normalisation)**.** *If* $\Gamma \vdash t : A$*, then* $\mathsf{SN}_\Gamma(t)$*. That is,* t *is strongly normalising under any substitution of strongly normalising terms for the variables in* Γ*.*

Proof. Let σ be an arbitrary substitution assigning to each $x : B$ in Γ a closed term $\sigma(x) \in [\![B]\!]$. By soundness of reducibility, we have $\sigma(t) \in [\![A]\!]$, and by CR1, $\sigma(t)$ is strongly normalising. Since σ was arbitrary, this establishes that t is strongly normalising under all strongly normalising substitutions. That is, $\mathsf{SN}_\Gamma(t)$. ■

5.1.2 Confluence

References

[Liu23] Liu, Kit. *Structural Foundations in Topoi*. University of Warwick. 2023.

[Sou21a] Southwell, Richard. *Topos Theory Essentials*. 2021. URL: https://www.youtube.com/watch?v=t7tLRUI1E-E.

[Sou21b] Southwell, Richard. *Internal Language of a Topos (Mitchell-Bénabou Language)*. 2021. URL: https://www.youtube.com/watch?v=66glvnHX-a8.

[HB34] Hilbert, David and Bernays, Paul. *Foundations of Mathematics Vol. 1*. 1934.

[Iem24] Iemhoff, Rosalie. "Intuitionism in the Philosophy of Mathematics". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Zalta, Edward N. and Nodelman, Uri. Summer 2024. Metaphysics Research Lab, Stanford University, 2024.

[Lin72] Linderholm, Carl E. *Mathematics made Difficult*. World Publishing, 1972.

[PB22] Pradic, Cécilia and Brown, Chad E. *Cantor-Bernstein implies Excluded Middle*. 2022. arXiv: 1904.09193 [math.LO]. URL: https://arxiv.org/abs/1904.09193.

[Dav05] Davies, E. Brian. "A Defence of Mathematical Pluralism". In: *Philosophia Mathematica* 13.3 (2005), pp. 252–276. DOI: 10.1093/philmat/nki017.

[Mil20] Mileti, Joseph R. *A Mathematical Introduction to Mathematical Logic*. 2020.

[Bab19] Babb, Ryan Acosta. *All Functions are Continuous! A provocative introduction to constructive analysis*. University of Warwick. 2019.

[Wei95] Weihrauch, Klaus. *A Simple Introduction to Computable Analysis*. Fernuniv., Fachbereich Informatik, 1995.

[Coq04] Coquand, Thierry. "About Brouwer's fan theorem". In: *Revue internationale de philosophie* 230 (2004), pp. 483–489.

[Bau11] Bauer, Andrej. *An injection from N^{N} to N*. 2011.

[FH06] Fourman, Michael P and Hyland, J Martin E. "Sheaf models for analysis". In: *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*. Springer. 2006, pp. 280–301.

[FS06] Fourman, Michael P and Scott, Dana S. "Sheaves and logic". In: *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*. Springer. 2006, pp. 302–401.

[Gra06] Grayson, Robin J. "Heyting-valued models for intuitionistic set theory". In: *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*. Springer. 2006, pp. 402–414.

[Lei24] Leinster, Tom. *Axiomatic Set Theory*. University of Edinburgh. 2024.

[Hin97] Hindley, J Roger. *Basic Simple Type Theory*. 42. Cambridge University Press, 1997.

[Mit72] Mitchell, William. "Boolean topoi and the theory of sets". In: *Journal of Pure and Applied Algebra* 2.3 (1972), pp. 261–274.

[Ber21] Berger, Josef. *Brouwer's fan theorem*. 2021. arXiv: 2001.00064 [math.LO]. URL: https://arxiv.org/abs/2001.00064.

[Van90] Van Stigt, Walter P. *Brouwer's Intuitionism*. North-Holland Amsterdam, 1990.

[Vel06] Veldman, Wim. *Brouwer's Real Thesis on Bars*. CS 6. Université Nancy 2, 2006.

[Jac99] Jacobs, Bart. *Categorical Logic and Type Theory*. Vol. 141. Elsevier, 1999.

[Shu16] Shulman, Michael. "Categorical logic from a categorical point of view". In: *Available on the web* 19 (2016), pp. 20–22.

[Ves79] Vesley, Richard. "Choice sequences. A chapter of intuitionistic mathematics. Oxford logic guides. Clarendon Press, Oxford1977, ix+ 170 pp." In: *The Journal of Symbolic Logic* 44.2 (1979), pp. 275–276.

[Tro+96] Troelstra, Anne Sjerp et al. "Choice sequences: a retrospect". In: *CWI Quarterly* 9 (1996), pp. 143–149.

[Bra16] Brattka, Vasco. "Computability and Analysis, a Historical Approach". In: *Pursuit of the Universal*. Springer International Publishing, 2016, pp. 45–57. ISBN: 9783319401898. DOI: 10.1007/978-3-319-40189-8_5.

[Bri99] Bridges, Douglas S. "Constructive mathematics: a foundation for computable analysis". In: *Theoretical computer science* 219.1-2 (1999), pp. 95–109.

[TD88] Troelstra, A.S. and Dalen, D. van. *Constructivism in Mathematics, Vol 1*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1988. ISBN: 9780080570884.

[TD14] Troelstra, A.S. and Dalen, D. van. *Constructivism in Mathematics, Vol 2*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2014. ISBN: 9780080955100.

[McL92] McLarty, Colin. *Elementary Categories, Elementary Toposes*. Clarendon Press, 1992.

[Dum00] Dummett, Michael. *Elements of Intuitionism*. Vol. 39. Oxford University Press, 2000.

[Bau17] Bauer, Andrej. "Five stages of accepting constructive mathematics". In: *Bulletin of the American Mathematical Society* 54.3 (2017), pp. 481–498.

[Som11] Sommaruga, Giovanni. *Foundational Theories of Classical and Constructive Mathematics*. Vol. 76. Springer Science & Business Media, 2011.

[Bis67] Bishop, Errett. *Foundations of Constructive Analysis*. 1967.

[Bee12] Beeson, Michael J. *Foundations of Constructive Mathematics: Metamathematical Studies*. Vol. 6. Springer Science & Business Media, 2012.

[Tro+11] Troelstra, Anne Sjerp et al. "History of constructivism in the 20th century". In: *Set Theory, Arithmetic, and Foundations of Mathematics* (2011), pp. 150–179.

[Lak63] Lakatos, Imre. *Proofs and Refutations*. Nelson London, 1963.

[Fra20] Frank, Matthew. "Interpolating between choices for the approximate intermediate value theorem". In: *Logical Methods in Computer Science* 16 (2020).

[Hen12] Hendtlass, Matthew. "The intermediate value theorem in constructive mathematics without choice". In: *Annals of Pure and Applied Logic* 163.8 (2012). Continuity, Computability, Constructivity: From Logic to Algorithms, pp. 1050–1056. ISSN: 0168-0072. DOI: https://doi.org/10.1016/j.apal.2011.12.026.

[Loo13] Loomis, Lynn H. *Introduction to Abstract Harmonic Analysis*. Courier Corporation, 2013.

[Str03] Streicher, Thomas. "Category Theory and Categorical Logic". In: *Lecture Notes, Technische Universität Darmstadt.* (2003).

[LS88] Lambek, Joachim and Scott, Philip J. *Introduction to Higher-order Categorical Logic*. Vol. 7. Cambridge University Press, 1988.

[Hey66] Heyting, Arend. *Intuitionism: An Introduction*. Vol. 41. Elsevier, 1966.

[MV20] Moschovakis, Joan R. and Vafeiadou, Garyfallia. *Intuitionistic Mathematics and Logic*. 2020. arXiv: 2003.01935 [math.LO]. URL: https://arxiv.org/abs/2003.01935.

[MML18] MARTINO., Enrico, Martino, Enrico, and Lue. *Intuitionistic Proof versus Classical Truth*. Springer, 2018.

[Bel14] Bell, John Lane. *Intuitionistic Set Theory*. Vol. 50. College Publications London, 2014.

[VS17] Van Atten, Mark and Sundholm, Göran. "LEJ Brouwer's 'Unreliability of the Logical Principles': A New Translation, with an Introduction". In: *History and Philosophy of Logic* 38.1 (2017), pp. 24–47.

[Acz11] Aczel, Peter. *Local Constructive Set Theory and Inductive Definitions*. Springer, 2011.

[SU06] Sørensen, Morten Heine and Urzyczyn, Pawel. *Lectures on the Curry-Howard Isomorphism*. Vol. 149. Elsevier, 2006.

[Avi22] Avigad, Jeremy. *Mathematical Logic and Computation.* Cambridge University Press, 2022.

[VV94] Van Dalen, Dirk and Van Dalen, Dirk. *Logic and structure.* Vol. 3. Springer, 1994.

[Tro73] Troelstra, Anne Sjerp. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Springer, 1973.

[Van05] Van Dalen, Dirk. *Mystic, Geometer, and Intuitionist: The Life of LEJ Brouwer.* Vol. 2. Oxford University Press, 2005.

[Bel] Bell, John L. *Notes on toposes and local set theories.*

[Nor] Norwood, Zach. *Tychonoff's Theorem Implies AC.*

[Van04] Van Atten, Mark. *On Brouwer.* Wadsworth, 2004.

[Mad92] Maddy, Penelope. *Realism in Mathematics.* Oxford University Press, 1992.

[Coq14] Coquand, Thierry. "Recursive Functions and Constructive Mathematics". In: *Constructivity and computability in historical and philosophical perspective* (2014), pp. 159–167.

[BV07] Bridges, Douglas S and Vita, Luminita Simona. *Techniques of Constructive Analysis.* Springer Science & Business Media, 2007.

[BH24] Bauer, Andrej and Hanson, James E. *The Countable Reals.* 2024. arXiv: `2404.01256 [math.LO]`. URL: `https://arxiv.org/abs/2404.01256`.

[Hyl82] Hyland, J Martin E. "The effective topos". In: *Studies in Logic and the Foundations of Mathematics.* Vol. 110. Elsevier, 1982, pp. 165–216.

[Car14] Caramello, Olivia. "Topos-theoretic background". In: (2014). URL: `http://www.oliviacaramello.com/Unification/ToposTheoreticPreliminariesOliviaCaramello.pdf`.

[Bel08] Bell, John L. *Toposes and Local Set Theories: An Introduction.* Courier Corporation, 2008.

[BW00] Barr, Michael and Wells, Charles. *Toposes, Triples, and Theories.* Springer-Verlag, 2000.

[Ble21] Blechschmidt, Ingo. *Using the internal language of toposes in algebraic geometry.* 2021. arXiv: `2111.03685 [math.AG]`. URL: `https://arxiv.org/abs/2111.03685`.

[BMW85] Barr, Michael, Mclarty, Colin, and Wells, Charles. "Variable set theory". In: *Unpublished manuscript* (1985).

[BR87] Bridges, Douglas and Richman, Fred. *Varieties of Constructive Mathematics.* Vol. 97. Cambridge University Press, 1987.

[Gol84] Goldblatt, Robert. *Topoi: The Categorial Analysis of Logic.* Elsevier, 1984. ISBN: 9780444867117.

[MM12] MacLane, S. and Moerdijk, I. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory.* Springer New York, 2012. ISBN: 9781461209270.

[Kos12] Kostecki, Ryszard. *An Introduction to Topos Theory.* 2012.

[Lei11] Leinster, Tom. *An informal introduction to topos theory.* 2011. arXiv: `1012.5647 [math.CT]`.

[Str04] Streicher, Thomas. *Universes in Toposes.* Clarendon Press, 2004. ISBN: 9780198566519.

[Men30] Menger, Karl. "Der Intuitionismus". In: *Blätter Für Deutsche Philosophie* 4 (1930), pp. 311–325.