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, we avoid 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 $\big[\varnothing \to \{\varnothing\}\big]$;
- 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. We 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:

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

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:

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

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 $f \coloneqq \lambda x.x + 1$ 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 simpler 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 $A \to B$ by *abstracting* over x.

- Elimination:
$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B}(\to_{\mathrm{E}})$$
Given a function $f : A \to B$, we can *apply* the function to an argument $a : A$ to obtain a term of type B. We can optionally add brackets to applications $f(a)$ to clarify arguments.

- Computation:
$$\frac{}{\Gamma \vdash (\lambda x.t)a \equiv t[a/x]}(\to_{\beta})$$
Moreover, applying a function abstraction to an argument reduces to the substitution of the argument for the bound variable in the body of the function.

- 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.*

Function types $A \to B$ syntactically internalise the semantic notion of a map from elements of A to elements of B. The introduction rule defines a function by *abstraction*: if we have a term t that is of type B in a context containing $x : A$, we can fold the dependency into a single term $\lambda x x.t$ to obtain a function of type $A \to B$. Function application is then realised syntactically by reducing $(\lambda x.t)a$ to $t[a/x]$, replacing the free variable x with the argument value a.

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:

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

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.

2.2.3 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.

Sum types $A + B$ represent data given by *alternatives*, where a term may arise from distinct sources, and computations must analyse which case has occurred. Unlike product types, which collect information from both components simultaneously, sum types express *branching* structure. They introduce a notion of data whose shape may vary, and whose use requires active inspection.

The key structural property of sum types is that they are defined entirely by partial information: when a term of type $A + B$ is given, we cannot access a component directly as with pairs; instead, we must determine which constructor was used.

In informal mathematics, these kinds of unions are typically disjoint only by convention, and distinguishing which component an element originated from may require manual tagging. Sum types make this distinction intrinsic; the constructors inl and inr enforce that every term carries canonical evidence of its origin, meaning that case analysis is always well-defined and complete.

Moreover, the elimination rule for sums captures an essential principle of mathematical definition: to define an object on a domain formed as a union, it suffices to define it on each constituent piece. In classical set theory, this is a theorem; in type theory, it is built into the structure of the type itself.

2.2.4 More Base Types

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

- the unit type;
- and the void 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 $b : B$:

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

- Computation: intuitively, since unit has only inhabitant, there is no information in a term $t : \mathsf{unit}$ to influence the choice of result – the provided term b is therefore the only reduct that can arise from eliminating the canonical inhabitant $\star$:

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

- 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 produce 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.

2.3 Derivations

Now that we have introduced some rules for introducing terms, we can combine them to construct more complex terms.

A *derivation* of a judgement is a finite tree whose nodes are labelled by judgements, such that each node is the conclusion of an instance of an inference rule whose premises label its immediate subtrees. The leaves are then instances of rules with no premises.

If there exists a such a tree, with root labelled by a judgement J, then we say that J is *derivable*. Thus, derivations are finite witnesses of the derivability of judgements, recording how inference rules combine to produce more complex terms and types from simpler ones.

Apart from the typing formation judgements we have seen so far, our 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. While this rule appears simple, it already allows the derivation of a rich set of judgements when combined with the rules for constructing terms of composite types.

Example. The term $\lambda x.\langle x,x\rangle : A \to A \times A$ is derivable:

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}(\text{axiom}) \qquad \dfrac{}{x : A \vdash x : A}(\text{axiom})}{x : A \vdash \langle x,x\rangle : A \times A}(\times_{\mathrm{I}})}{\lambda x.\langle x,x\rangle : A \to A \times A}(\to_{\mathrm{I}})$$

△

We should emphasise that all of the inference rules that we have seen so far have been rule *schemata*, ranging over all admissible substitutions of contexts, terms, and types. For example, the product introduction rule

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a,b\rangle : A \times B}$$

is a schema where each choice of sequents $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ yields a particular *instance* of the rule. This distinction between rule schemata (meta-level templates) and their instances (object-level applications) is important when we reason about derivations and when we prove admissibility of additional rules.

2.3.1 Admissible Rules

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 – any derivation using an admissible rule can be reconstructed without it.

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 $\Gamma,\Delta \vdash x : A$. Throughout the proof, assume that y is fresh for Γ,Δ; if any binder in a subderivation would capture y, we can first α-convert that binder so that it does not.

- If the last rule is the axiom rule, then 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 $A \equiv C \to D$):

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

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

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

  and abstraction again yields

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

  as required.

- Suppose the last rule is application:

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

  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.

■

Most other similar results proceed similarly by induction on derivations, using the fact that each inference rule is a schema whose instances behave uniformly under context extension, substitution, etc.

More generally, we can prove that a property P of judgements holds for all derivations in the same way, by proving that for every primitive rule schema, whenever P holds for all premises, P also holds of the conclusion.

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

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

  By repeated applications of this rule, contexts may freely permute all typing declarations.

- 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.

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

- The *substitution rule* says that if two variables in the context share the same type, then all occurrences of one may be replaced by the other, thereby contracting the context by identifying the two assumptions:

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

- The *cut rule* says that if a term of type A can be derived in one context and a term of type B can be derived in another context assuming a variable of type A, then the latter derivation can be completed by substituting the first term for the assumption, yielding a derivation of B in the combined context:

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

  This "cuts out" the intermediate derivation of A.

Given this flexibility of contexts, one may wonder why we define contexts as *lists*, rather than *sets* (in the sense that ordering and multiplicities don't affect derivability). Firstly, some of these rules will fail once we move to a more expressive type theory, where context variables may start to depend on one another.

But more importantly, proofs and typing derivations are defined inductively; it is much simpler to define an inductive relation over lists than over sets, because lists have canonical constructors: a list is either the empty list, or is a typing declaration glued to a smaller list.

$$\cdot\ \mathsf{ctx} \qquad\qquad \Gamma, x : A\ \mathsf{ctx}$$

In contrast, reasoning inductively on finite sets is awkward, requiring arbitrary choices of elements, or including additional reasoning about permutations.

These structural rules can also fail in more *restrictive* type theories, so the list-based representation is the more general one; and the set-like behaviour of our simple type theory is recovered as a theorem.

Theorem 2.2. *The exchange, contraction, substitution, and cut rules are all admissible.*

Proof. By structural induction on derivations. ■

Having established the syntactic foundations of the type theory and shown that the structural rules are admissible, we have now completed the purely derivational development of the system. In particular, the preceding results guarantee that derivability is stable under standard structural transformations of contexts and that substitution behaves coherently with respect to derivations. These results ensure that the proof-theoretic framework of the type theory is well behaved, in the sense that derivations depend only on the essential connective structure of the rules – how premises are related to conclusions – rather than on the particular syntactic details of how inferences are presented. These results belong to the *static* syntactic analysis of the theory.

We now transition to a complementary view, studying the *operational* semantics of the theory. We will see that judgemental equality carries not only a syntactic meaning with respect to derivations, but also a *computational* one with respect to *reduction*.

2.4 Term Reduction

This section develops certain metatheoretic properties of the *reduction* and judgemental equality relations. The results here will be useful in proving later results, 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 purely mathematical uses of 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.

The extrinsic perspective is convenient here because the results covered in this section apply to more general *rewriting systems* than the one induced by our type theory, and their proofs only rely on reasoning about arbitrary syntactic terms, independently from any notion of typing. However, once these metatheoretic results are established, they will be applied to well-typed terms in the usual, intrinsically typed setting.

2.4.1 Operational Semantics of Judgemental Equality

So far, we have treated definitional equality $\equiv$ of terms as a primitive judgement, equipped with the standard inference rules for an equivalence relation (reflexivity, symmetry, and transitivity), together with stability under substitution and α-conversion. These rules determine the notion of equality in terms of the *derivational* content of the theory.

However, we can also interpret the β rules as a system for rewriting or *reducing* terms into simpler forms, thus expressing a form of *computation*. From this perspective, equality can be defined *operationally* in terms of this reduction system, where two terms are judgementally equal if they compute to a common term.

In a well-behaved rewriting system – one in which reduction is *normalising* 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 *normalising*, 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 form* 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 under substitution ("*congruence of reduction*"):

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

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 $t \longrightarrow t'$. We also abbreviate "t is a term in normal form" to "t is a normal term".

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

The one-step reduction relation $\longrightarrow$ defines precisely how computation progresses in our system. A normal form is then a term which cannot be further reduced under any of the β rules, either at the top level or within subterms – it is the final result of a computation.

While the form of normal terms of base types is straightforward, it is not immediately obvious what the normal form of a term of a compound type looks like. Applications, projections, and case analyses can all produce reducible expressions, so determining which forms are irreducible requires a careful syntactic analysis. In particular, it is important to distinguish between terms that are syntactically normal (i.e., no reduction applies) and terms that are canonical with respect to their type. While all canonical terms are normal, not all normal terms are canonical in a naïve sense, because they may contain subterms whose form could still be "simplified" if they were not closed or if they were reducible.

Lemma 2.3 (Canonical Forms)**.** *Let $t : T$ be a closed term in normal form. Then,*

(*i*) *If T is a base type, then t is a canonical constant given by the type's constructor(s);*

(*ii*) *If $T \equiv A \to B$, then $t \equiv \lambda x.u$ for some $x : A$ and $u : B$;*

(*iii*) *If $T \equiv A \times B$, then $t \equiv \langle a,b\rangle$ for some $a : A$ and $b : B$;*

(*iv*) *If* $T \equiv A + B$*, then either* $t \equiv \mathsf{inl}(a)$ *for some* $a : A$*, or* $t \equiv \mathsf{inr}(b)$ *for some* $b : B$*.*

and moreover, all the closed subterms in t *are also in canonical forms.*

Proof. By induction on the derivation of $t : T$. By induction hypothesis, all closed subterms of t are already in canonical form. The proof then proceeds by case analysis on the type T.

(*i*) By the typing rules of base types, t is either:

- a constant of that type (e.g. $\star : \mathsf{unit}$);
- or a variable.

and no other inference rules produce a base type. Since t is closed, it cannot contain a variable, so t must be a canonical constant of T.

(*ii*) By the syntax of our theory, terms of function types can be of the form of:

- (axiom) a variable $x : A \to B$;
- ($\to_{\mathrm{I}}$) a λ-abstraction $\lambda x.u$;
- ($\to_{\mathrm{E}}$) an application $u\,v$ where $u : C \to (A \to B)$ and $v : C$.
- ($\times_{\mathrm{E}}$) a projection $\mathsf{fst}(x)$ with $x : (A \to B) \times C$, or $\mathsf{snd}(y)$ with $y : C \times (A \to B)$;
- ($+_{\mathrm{E}}$) a case analysis $\mathsf{case}(x; f,g)$ with $x : C + D$, $f : C \to A \to B$, and $g : D \to A \to B$.

The other inference rules produce a type with a different top-level syntax.

By assumption, t is closed, ruling out axiom. In the application case, $u\,v$ is either a redex, or u is a variable. In either case, this contradicts that t is normal or closed, respectively. Similarly, for product projections, x is either a pair or x is a variable; and for case analyses, x is either an injection or a variable; again, in either case, this contradicts that t is normal or closed, respectively.

The only remaining possibility is $t \equiv \lambda x.u$. Any closed subterms of u (i.e. those not involving x) must be in normal form, so by the inductive hypothesis, they also satisfy canonical forms, and hence $t \equiv \lambda x.u$ with canonical subterms.

(*iii*) Terms of a product type can be of the form of:

- (axiom) a variable $x : A \times B$;
- ($\times_{\mathrm{I}}$) a pair $\langle a,b\rangle$;
- ($\times_{\mathrm{E}}$) a projection $\mathsf{fst}(x)$ with $x : (A \times B) \times C$, or $\mathsf{snd}(y)$ with $y : C \times (A \times B)$;
- ($\to_{\mathrm{E}}$) an application $u\,v$ where $u : C \to (A \times B)$ and $v : C$.
- ($+_{\mathrm{E}}$) a case analysis $\mathsf{case}(x; f,g)$ with $x : C + D$, $f : C \to (A \times B)$, and $g : D \to (A \times B)$.

The first and last three cases are excluded as above, so we must have $t \equiv \langle a,b\rangle$. Since $a : A$ and $b : B$ are normal closed subterms, they are also in canonical forms by the induction hypothesis.

(*iv*) Terms of a sum type can be of the form of:

- (axiom) a variable $x : A + B$;
- ($+_{\mathrm{I}}$) an injection $\mathsf{inl}(a)$ with $a : A$, or $\mathsf{inr}(b)$ with $b : B$;
- ($\to_{\mathrm{E}}$) an application $u\,v$ where $u : C \to (A + B)$ and $v : C$.
- ($\times_{\mathrm{E}}$) a projection $\mathsf{fst}(x)$ with $x : (A + B) \times C$, or $\mathsf{snd}(y)$ with $y : C \times (A + B)$;
- ($+_{\mathrm{E}}$) a case analysis $\mathsf{case}(x; f,g)$ with $x : C + D$, $f : C \to (A + B)$, and $g : D \to (A + B)$.

The first and last three cases are excluded as above, so we must have $t \equiv \mathsf{inl}(a)$ or $t \equiv \mathsf{inr}(b)$. Again, $a : A$ and $b : B$ are normal closed subterms, so they are also in canonical forms by the induction hypothesis.

■

One immediate consequence of this characterisation is that it ensures safe and complete destructibility of closed terms by the appropriate eliminators: it is always possible to β-reduce a redex formed from a closed normal term.

For instance, if we know that a closed term $t : A \times B$ is in normal form, the lemma guarantees that t must be a pair $\langle a,b \rangle$ whose components are also similarly canonical. This means that applying the standard destructors fst and snd will always succeed in extracting meaningful subterms, rather than producing a "stuck" term that cannot be reduced further.

Similarly, for sum types $A + B$, a closed term in normal form must be either $\mathsf{inl}(a)$ or $\mathsf{inr}(b)$, so a case analysis over the term will always be exhaustive and will never encounter an ill-formed argument; while for function types, any closed term of type $A \to B$ is guaranteed to be a λ-abstraction, ensuring that function application is always defined when applied to an argument in the domain type.

In the case of base types, the lemma confirms that the only closed normal terms are canonical constants, reinforcing the idea that such terms are fully evaluated and irreducible.

2.4.2 Strong Normalisation

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 $t \longrightarrow^* t''$.

A *closed* term t is *weakly normalising* if it has a reduction path to a normal form. That is, there exists a normal term t' with

$$t \longrightarrow^* t'$$

Example. The term

$$\Omega :\equiv (\lambda x.xx)(\lambda x.xx)$$

is not even weakly normalising, since it is its own β-reduct:

$$\Omega \equiv (\lambda x.x\,x)(\lambda x.x\,x) \longrightarrow (\lambda x.x\,x)(\lambda x.x\,x) \equiv \Omega$$

Conversely, the term

$$(\lambda y.\star)\,\Omega$$

is weakly normalising, since applying β-reduction to the outer abstraction yields the finite reduction sequence

$$(\lambda y.\star)\,\Omega \longrightarrow \star$$

Note, however, the possibility of infinitely reducing Ω is still open:

$$(\lambda y.\star)\,\Omega \longrightarrow (\lambda y.\star)\,\Omega \longrightarrow (\lambda y.\star)\,\Omega \longrightarrow \cdots$$

△

As illustrated by this example, different choices of reductions may produce different results on weakly normalising terms.

An algorithm by which these reductions are selected is called a *reduction strategy*. These are well-studied in computer science, because the choice of strategy can directly affect both the termination and *efficiency* of computations, making the analysis of evaluation order central to both theoretical and practical aspects of programming language semantics. Some standard strategies include:

- *Normal-order reduction*: always reduce the leftmost, outermost redex first. For practical applications, this strategy can be rather inefficient, as evaluating outermost redexes can often duplicate subterms, increasing the number of reductions needed, and for complex terms, identifying the outermost redex in the first place can also be non-trivial. Deferred reductions can also cause additional memory overhead as deferred computations ("*thunks*") also need to be stored.
- *Applicative-order reduction*: reduce the leftmost, innermost redex first. This corresponds to the evaluation strategy of most eager functional languages like OCaml. While often more efficient in practice because it avoids repeating work and never generates any additional thunks, it also always evaluates all function arguments, even if they aren't used.
- *Call-by-name*: similar to normal-order reduction but restricted to function arguments; arguments are substituted without being evaluated first. In functional languages, this can be desirable, as the complex functions commonly used in such languages often manipulate other functions or delayed computations. Lazy evaluation also allows for infinite structures like streams and infinite trees, since operations can traverse only the part of the structure that is needed, leaving the rest as a thunk. Haskell uses an optimised version of this strategy, where evaluated arguments are memoised, so that subsequent uses reuse the result, rather than recomputing it.
- *Call-by-value*: only reduce function arguments before substitution. This strategy corresponds to *strict evaluation* in most programming languages. Call-by-value can be more efficient than call-by-name in avoiding repeated computation but does not guarantee termination for all weakly normalising terms.

Despite its practical inefficiencies, the normal-order reduction strategy has important theoretical properties: if a term is weakly normalising, normal-order reduction will eventually reach a normal form. In contrast, the applicative-order and call-by-value strategies may fail to normalise a weakly normalising term, as demonstrated by $(\lambda y.\star)\Omega$.

The key reason normal-order reduction is normalising is that it always delays the evaluation of potentially non-terminating arguments until their values are actually needed. This ensures that computations that do not contribute to the final normal form are never performed, allowing the reduction to terminate even in the presence of divergent subterms. In contrast, applicative-order and call-by-value strategies evaluate arguments eagerly, which can force the evaluation of expressions that are ultimately irrelevant to the final result, leading to non-termination.

These strategies highlight an important trade-off in functional programming languages: delaying evaluation increases the chance of termination and avoids unnecessary computation, but at the cost of potential repeated reductions or increased memory usage when unevaluated expressions are duplicated.

Although weak normalisation ensures that a term admits some terminating reduction sequence, this property is too fragile to support deeper metatheoretic reasoning. The mere existence of a normal form does not guarantee that it can be reached by an arbitrary reduction strategy, and certain sequences of reductions may diverge indefinitely; and establishing that normal-order reduction always reaches a normal form for weakly normalising terms is technically involved, relying on detailed syntactic arguments about the order of redex contractions,

Reasoning along these lines can quickly become cumbersome in developing later metatheoretic results – it would be preferable to avoid dependence on any specific reduction strategy altogether. For our uses, we will instead distinguish between terms that merely *can* terminate and those for which termination is *inevitable*, regardless of the choices made during reduction.

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

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

Equivalently, every reduction sequence terminates in finitely many steps.

Lemma 2.4. *If t is strongly normalising, then all of its one-step reducts $t \longrightarrow t'$ are also strongly normalising.*

Proof. If t' were not, then there exists an infinite reduction sequence starting from t'. But then, prepending $t \to t'$ to this sequence yields an infinite reduction sequence starting at t, contradicting that t is strongly normalising. ■

Because reduction is defined purely syntactically, both of these definitions of normalisation also make sense for open terms. However, these are both relatively fragile notions, as they are not closed under substitution: weakly/strongly normalising open terms may begin to diverge once their free variables are instantiated by actual terms:

Example. The open term x consisting of a single variable is trivially strongly normalising in this syntactic sense, but is not even weakly normalising if we instantiate x as Ω. △

For metatheoretic analysis, we usually require a stronger, substitution-closed notion of strong normalisation for open terms.

A term t in context $\Gamma \equiv x_i : A_i, \ldots, x_n : A_n$ is *strongly normalising* if for every substitution σ mapping each variable $x_i : A_i \in \Gamma$ to a strongly normalising closed term $\sigma(x_i)$, the term $t[\sigma]$ obtained by applying σ to all the free variables of t is strongly normalising.

The requirement that the substituted term is strongly normalising in this definition is to avoid trivial non-termination arising purely from a divergent term being substituted in, ensuring that strong normalisation is a property of the term itself, and not on the terms it might be instantiated with.

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

Theorem 2.5 (Strong Normalisation)**.** *Every well-typed term is strongly normalising. That is, if $\Gamma \vdash t : A$, then $\mathsf{SN}_\Gamma(t)$.*

It turns out that for the typed fragment, *every* term is strongly normalising; our choice of evaluation strategy does not affect the termination of a computation (though they may take different amounts of time and memory to get there).

A direct consequence of this property is that non-normalising terms like the Ω combinator are not typeable. In untyped λ calculus, Ω is the canonical example of a term with no normal form, exhibiting infinite self-application – but such a term is prohibited at the syntactic level by types.

The fact that such terms are untypable in our system illustrates how the type system enforces a close alignment between syntactic well-formedness and computational soundness, ensuring that all well-typed terms are meaningful in a precise *constructive* sense.

The proof strategy for this result is standard, following the *reducibility method*. Before we proceed with the proof of strong normalisation, we first provide some auxiliary definitions and results.

For each type A, we define the set $[\![A]\!]$ of strongly normalising closed terms of type A, called *reducibility candidates* of A. We call elements of these sets *reducible* terms. These sets are defined via structural recursion on A as follows:

- Base types A (including unit and void, though the latter only vacuously):

  $$[\![A]\!] \coloneqq \{t : A \mid \mathsf{SN}(t)\}$$

  as all basic terms cannot reduce further;

- Function types $A \to B$:

$$[\![A \to B]\!] \coloneqq \{f : A \to B \mid \mathsf{SN}(f) \land \forall a \in [\![A]\!].\, f\, a \in [\![B]\!]\}$$

  so that t behaves well on all reducible arguments:

- Product types $A \times B$:

$$[\![A \times B]\!] \coloneqq \{t : A \times B \mid \mathsf{SN}(t) \land \mathsf{fst}(t) \in [\![A]\!] \land \mathsf{snd}(t) \in [\![B]\!]\}$$

  so that components are reducible;

- Sum types $A + B$:

$$[\![A + B]\!] \coloneqq \{t : A + B \mid \mathsf{SN}(t) \land \forall C\ \mathsf{type}.\forall f \in [\![A \to C]\!].\forall g \in [\![B \to C]\!].\mathsf{case}(t; f,g) \in [\![C]\!]\}$$

  so that case analysis on reducible branches yields reducible results.

The idea is that, instead of reasoning directly about arbitrary well-typed terms, we restrict attention to terms that are already strongly normalising and verify that they behave well under the type's operations. This recursive characterisation ensures that the notion of reducibility is closed under the structure of types.

Lemma 2.6 (Closure under Reduction)**.** *If $t \in [\![A]\!]$ and $t \longrightarrow t'$, then $t' \in [\![A]\!]$.*

Proof. By induction on the type A. Note that if t is strongly normalising, then so is t'.

- Base case: if A is a base type, then this is trivial.
- Functions $A \to B$: for all $a \in [\![A]\!]$, we have by congruence of reduction:

$$t\, a \to t'\, a$$

  By the inductive hypothesis on B (B is closed under one-step reductions), we have $t'a$, and hence $t' \in [\![A \to B]\!]$.

- Products $A \times B$: by congruence of reduction:

$$\mathsf{fst}(t) \longrightarrow \mathsf{fst}(t'), \qquad \mathsf{snd}(t) \longrightarrow \mathsf{snd}(t')$$

  By the inductive hypothesis on A and B, $\mathsf{fst}(t') \in [\![A]\!]$ and $\mathsf{snd}(t') \in [\![B]\!]$, and hence $t' \in [\![A \times B]\!]$.

- Sums $A + B$: for all $f : A \to C$ and $g : B \to C$, we have by the congruence of reduction:

$$\mathsf{case}(t; f,g) \longrightarrow \mathsf{case}(t'; f,g)$$

  By the inductive hypothesis on C, we have $\mathsf{case}(t'; f,g) \in [\![C]\!]$ and hence $t' \in [\![A + B]\!]$

■

By induction on the length of reduction sequences, this result establishes that reducibility is stable under arbitrary computations. For compound types, this closure interacts naturally with the type structure: for a reducible function, applying it to a reducible argument yields a term whose reducts remain reducible; while for pairs or sums, evaluating a component or performing a case analysis cannot lead outside the corresponding reducibility candidate. In this way, closure under reduction guarantees that reducible terms are stable under the operational semantics of our language, and are hence well-behaved with respect to all type-directed computations.

The next lemma expands on this, lifting reducibility to *open* terms through reducible substitutions:

Lemma 2.7 (Soundness of Reducibility)**.** *If $\Gamma \vdash t : A$, and σ is any substitution assigning to each $x_i : A_i$ declared in Γ a term $\sigma(x_i) \in [\![A_i]\!]$, then the closed term $t[\sigma]$ belongs to $[\![A]\!]$.*

Proof. By induction on the typing derivation of t:

- Axiom $t \equiv x_i : A_i$. By assumption, $t[\sigma] = \sigma(x_i) \in [\![A_i]\!]$.
- $(\to_{\mathrm{I}})$ $t \equiv \lambda x.u : A \to B$. Let $a \in [\![A]\!]$. Then, $(\lambda x.u)[\sigma]\, a \longrightarrow u[\sigma,a/x]$. The reduct is closed, so by the inductive hypothesis applied to (the derivation of) the premise $\Gamma, x : A \vdash u : B$ (with substitution $\sigma, x \mapsto a$, which assigns reducible closed terms to all variables), we have $u[\sigma,a/x] \in [\![B]\!]$. As $a \in [\![A]\!]$ was arbitrary, we have $(\lambda x.u)[\sigma] \equiv t[\sigma] \in [\![A \to B]\!]$.
- $(\to_{\mathrm{E}})$ $t \equiv u\, v : B$. By the inductive hypothesis on the premises $\Gamma \vdash u : A \to B$ and $\Gamma \vdash v : A$, we have $u[\sigma] \in [\![A \to B]\!]$ and $v[\sigma] \in [\![A]\!]$. By the function clause, $u[\sigma]\, v[\sigma] \equiv t[\sigma] \in [\![B]\!]$.
- $(\times_{\mathrm{I}})$ $t \equiv \langle u,v\rangle : A \times B$. By the inductive hypothesis on $\Gamma \vdash u : A$ and $\Gamma \vdash v : B$, we have $u[\sigma] \in [\![A]\!]$ and $v[\sigma] \in [\![B]\!]$. Since the components are strongly normalising and the pair constructor doesn't introduce infinite reductions, $\langle u,v\rangle[\sigma] \equiv \big\langle u[\sigma],v[\sigma]\big\rangle$ is strongly normalising.

  Since reducibility sets are closed under reduction, the computation rules $\times_\beta$ yield $\mathsf{fst}\big\langle u[\sigma],v[\sigma]\big\rangle \equiv u[\sigma] \in [\![A]\!]$ and $\mathsf{snd}\big\langle u[\sigma],v[\sigma]\big\rangle \equiv v[\sigma] \in [\![B]\!]$, so $\big\langle u[\sigma],v[\sigma]\big\rangle \equiv \langle u,v\rangle[\sigma] \equiv t[\sigma] \in [\![A \times B]\!]$.
- $(\times_{\mathrm{E}})$ $t \equiv \mathsf{fst}(u) : A$ (the proof for $t \equiv \mathsf{snd}(v) : B$ is analogous). By the inductive hypothesis on $\Gamma \vdash u : A \times B$, we have $u[\sigma] \in [\![A]\!]$, so $\mathsf{fst}\big(u[\sigma]\big) \equiv \mathsf{fst}(u)[\sigma] \equiv t[\sigma] \in [\![A]\!]$.
- $(+_{\mathrm{I}})$ $t \equiv \mathsf{inl}(u) : A + B$ (the proof for $t \equiv \mathsf{inr}(v)$ is analogous). By the inductive hypothesis on $\Gamma \vdash u : A$, we have $u[\sigma] \in [\![A]\!]$, so $\mathsf{inl}\big(u[\sigma]\big) \equiv t[\sigma]$ is strongly normalising.

  Let $f \in [\![A \to C]\!]$ and $g \in [\![B \to C]\!]$. Then, $\mathsf{case}\big(\mathsf{inl}(u[\sigma]); f,g\big) \in [\![C]\!]$. By $+_\beta$, $\mathsf{case}\big(\mathsf{inl}(u); f,g\big) \equiv f\big(u[\sigma]\big)$, and since $f \in [\![A \to C]\!]$ and $u[\sigma] \in [\![A]\!]$, we have $f\big(u[\sigma]\big) \in [\![C]\!]$. As $f \in [\![A \to C]\!]$ and $g \in [\![B \to C]\!]$ were arbitrary, we have $\mathsf{inl}(u)[\sigma] \equiv t[\sigma] \in [\![A + B]\!]$.
- $(+_{\mathrm{E}})$ $t \equiv \mathsf{case}(u;\, \lambda x.v, \lambda y.w) : C$. By the inductive hypothesis on the three premises $\Gamma \vdash u : A + B$, $\Gamma \vdash f : A \to C$, and $\Gamma \vdash g : B \to C$, we have $u[\sigma] \in [\![A + B]\!]$, $f[\sigma] \in [\![A \to C]\!]$, and $g[\sigma] \in [\![B \to C]\!]$, respectively.

  By the sum clause, $\mathsf{case}\big(u[\sigma]; f[\sigma],g[\sigma]\big) \equiv \mathsf{case}(u; f,g)[\sigma] \equiv t[\sigma] \in [\![C]\!]$.

■

This result, often also referred to as the *fundamental lemma*, states that typing derivations are respected by the reducibility interpretation of types. In particular, substituting variables with reducible terms produces a term that remains reducible, ensuring that well-typed terms behave correctly with respect to evaluation, and that the typing system is sound relative to reducibility.

In particular, closed terms have no variables by definition, so the empty substitution from the empty context satisfies the hypotheses of the soundness lemma, immediately implying that every well-typed closed term is strongly normalising.

This connection allows us to deduce strong normalisation directly from the soundness lemma; since reducibility candidates contain only strongly normalising terms, any well-typed term belongs to the appropriate reducibility set when instantiated with any reducible substitution. This observation provides the key link needed to establish strong normalisation for all well-typed terms.

Theorem 2.5 (Strong Normalisation)**.** *Every well-typed term is strongly normalising. That is, if* $\Gamma \vdash t : A$*, then* $\mathsf{SN}_\Gamma(t)$.

Proof. Let σ be an arbitrary substitution assigning to each $x_i : A_i$ declared in Γ a term $\sigma(x_i) \in [\![A_i]\!]$. By the soundness of reducibility, we have $t[\sigma] \in [\![A]\!]$. By the definition of reducibility candidates, every term in $[\![A]\!]$ is strongly normalising, so $\mathsf{SN}\big(t[\sigma]\big)$. As σ was arbitrary, $\mathsf{SN}_\Gamma(t)$. ■

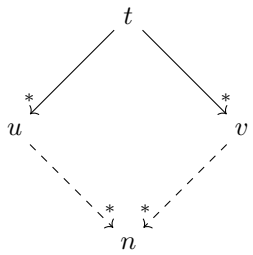
Strong normalisation implies that all computations in the system terminate. That is, every reduction sequence eventually reaches a normal form in a finite number of steps. This gives us inherent guarantee against infinite loops or non-terminating evaluations, which is a crucial property for both theoretical

analysis and practical implementation. Any function definable in the system is total, as every term representing a computation is guaranteed to produce a result.

2.4.3 Confluence

While strong normalisation ensures that every reduction sequence terminates, it does not by itself guarantee that this process is *deterministic*. It may be the case that distinct reduction paths from a common term could converge to distinct normal forms. In such a situation, although every computation eventually halts, its outcome may depend on the order in which reductions are applied. This ambiguity would undermine our identification of judgemental equality with reductive equality; if the same term could be reduced to two inequivalent results, then the notion of computation or proof normalisation would lose its intended meaning.

A system is *confluent* if, whenever a term t has reduction sequences to u and v, then there exist reduction sequences from u and v to some common term n:



From a broader perspective, confluence guarantees the coherence of the reduction system. It establishes that, although the operational semantics may allow for locally distinct reductions at any given point, all of these paths ultimately describe the same global computational behaviour. In other words, confluence formalises the idea that computation in type theory is consistent: regardless of how we proceed to simplify a term, the end result will be unique.

This property is particularly important for the model- and proof-theoretic aspects of type theory, since it ensures that normal forms are canonical representatives of equivalence classes under reduction, and that definitional equality corresponds to a single, well-determined notion of computation.

Moreover, the combination of strong normalisation and confluence implies the uniqueness of normal forms: normal reduction constitutes a computable total function from well-typed terms to their normal forms, rather than a mere relation.

The proof of confluence proceeds independently of strong normalisation, although the latter often simplifies certain arguments by precluding the existence of infinite reduction sequences.

The *parallel reduction* relation $\longrightarrow^{\parallel}$ is the smallest binary relation on terms such that

- $x \longrightarrow^{\parallel} x$ for all variables x;
- Abstraction:
$$\frac{t \longrightarrow^{\parallel} t'}{\lambda x.t \longrightarrow^{\parallel} \lambda x.t'}$$
- Application:
$$\frac{t \longrightarrow^{\parallel} t' \qquad u \longrightarrow^{\parallel} u'}{t\,u \longrightarrow^{\parallel} t'\,u'}$$
- Pairing:
$$\frac{t \longrightarrow^{\parallel} t' \qquad u \longrightarrow^{\parallel} u'}{\langle t, u\rangle \longrightarrow^{\parallel} \langle t', u'\rangle}$$
- Projection:
$$\frac{t \longrightarrow^{\parallel} t'}{\mathsf{fst}(t) \longrightarrow^{\parallel} \mathsf{fst}(t')} \qquad \frac{t \longrightarrow^{\parallel} t'}{\mathsf{snd}(t) \longrightarrow^{\parallel} \mathsf{snd}(t')}$$
- Injection:
$$\frac{t \longrightarrow^{\parallel} t'}{\mathsf{inl}(t) \longrightarrow^{\parallel} \mathsf{inl}(t')} \qquad \frac{t \longrightarrow^{\parallel} t'}{\mathsf{inr}(t) \longrightarrow^{\parallel} \mathsf{inr}(t')}$$
- Case analysis:
$$\frac{t \longrightarrow^{\parallel} t' \qquad f \longrightarrow^{\parallel} f' \qquad g \longrightarrow^{\parallel} g'}{\mathsf{case}(t; f, g) \longrightarrow^{\parallel} \mathsf{case}(t'; f', g')}$$

That is, every redex is reduced *simultaneously*.

Lemma 2.8 (Parallel Reductions)**.** *If $t \longrightarrow\!\!\!\!\!\rightarrow t'$, then $t \longrightarrow^{*} t'$.*

Theorem 2.9 (Diamond Property)**.** *If*

Theorem 2.10 (Confluence)**.**

In particular, confluence and strong normalisation provide a basis for the decidability of *type checking*: given a term and a type, we can systematically perform reductions until a normal form is reached and then inspect its structure, using the canonical forms lemma to compare the syntactic shape of the normal form against the expected form for the type in question. Since strong normalisation and confluence guarantees termination and determinacy, respectively, this yields a finite algorithm to determine whether a term inhabits a given type.

Similarly, type *inference* is also decidable: given a term, we can compute its normal form and identify its type just by inspecting the syntactic structure of this canonical representative.

2.4.4 Type Checking

3 The Curry-Howard Correspondence

Up to this point, our attention has been directed toward the syntactic and structural properties of our simple type theory; wwe have established rules describing how terms are introduced and eliminated, and how existing terms can be rewritten according to β- and η-reduction rules.

However, from this purely syntactic system, a striking phenomenon emerges – the rules we have formulated, though initially concerned only with the manipulation of formal expressions, display a compositional patterns that are reminiscent of *semantic reasoning*.

3.1 Term Erasure

All of the introduction and elimination rules we have seen so far do not depend on *term structure* in any way, in the sense that they inspect only the *types* of the premises. For example, in the application rule:

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}(\to_{\mathrm{E}})$$

we only require a value typed by the function type $A \to B$, and an argument typed by the domain A. In this rule, we do not require any knowledge of the internal structure of t or u.

For introduction and elimination rules, terms are treated as opaque objects of the appropriate type, and it is only when we look at the computation and uniqueness rules do we require analysing the term structure.

Following this observation, it is instructive to abstract away from term-level specifics, and analyse the essential *shape* of these introduction and elimination rules at the type level. If we ignore this term-level information, the function elimination rule reads as:

$$\frac{\Gamma \vdash A \to B, \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

This is precisely the natural deduction rule *modus ponens* for *implication* elimination: A implies B, and A holds, therefore B.

If we look at the introduction rule, the type information reads as:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

Again, this is precisely the natural deduction rule for implication introduction: if under the assumption A, we have B, then we can discharge the assumption A to obtain the implication $A \Rightarrow B$.

This syntactic identification is not a coincidence just for function types and implications; this process can be performed on every type former we have seen so far:

- Products $\times$ correspond to conjunction $\wedge$:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a,b\rangle : A \times B}(\times_{\mathrm{I}}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}(\wedge_{\mathrm{I}})$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{fst}(p) : A}(\times_{\mathrm{E}_1}) \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{snd}(p) : B}(\times_{\mathrm{E}_2}) \qquad \rightsquigarrow \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}(\wedge_{\mathrm{E}_1}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}(\wedge_{\mathrm{E}_2})$$

- Sums $+$ correspond to disjunction $\vee$:

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

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

- The unit type corresponds to tautology $\top$:

$$\frac{}{\Gamma \vdash t : \mathsf{unit}}(\mathsf{unit}_{\mathrm{I}}) \qquad \rightsquigarrow \qquad \frac{}{\Gamma \vdash \top}(\top_{\mathrm{I}})$$

$$\frac{\Gamma \vdash t : \mathsf{unit} \qquad \Gamma \vdash b : B}{\Gamma \vdash B}(\mathsf{unit}_{\mathrm{E}}) \qquad \rightsquigarrow \qquad \frac{\Gamma \vdash \top \qquad \Gamma \vdash B}{\Gamma \vdash B}(\top_{\mathrm{E}})$$

  In natural deduction, $\top$ is not usually given a elimination rule, because there is no information to extract from a tautologically true statement. The rule given here is a theorem, corresponding to this computational reading – the assumption $\top$ carries no data, and does not contribute anything in the proof of B.

- The void type corresponds to contradiction $\bot$:

$$\frac{\Gamma \vdash t : \mathsf{void}}{\Gamma \vdash B}(\mathsf{void}_{\mathrm{E}}) \qquad \rightsquigarrow \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash B}(\bot_{\mathrm{E}})$$

  The elimination rule here corresponds to the *principle of explosion*: from a contradiction, we can prove any statement B.

Under this procedure of erasing types, we see that our simple type theory encodes the inference rules of *intuitionistic propositional logic*.

As is standard in intuitionistic logic, we can define the negation operator by:

$$\neg A \equiv A \to \bot$$

This identification of types with *propositions* is the core observation of the *Curry–Howard correspondence*.

Theorem 3.1 (Term Erasure)**.** *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. Erasing term information from each of the introduction and elimination rules yields a valid inference rule in intuitionistic natural deduction, so by structural induction on derivations, the term erasure of a derivation π of a sequent $\Gamma \vdash t : A$ is a valid ■

Theorem 3.2. *If a type system satisfies strong normalisation, then it is logically consistent.*

Proof. If there were a closed term $t : \mathsf{void}$, then it reduces to a canonical form t'. Since reduction preserves type assignments, $t' : \mathsf{void}$. However, void does not have any constructors, so there is no canonical form of type void. ■

3.2 Type Erasure

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.

In fact, there is a remarkable correspondence, known as the *Curry–Howard isomorphism*, between proofs in logic and programs in type theory:

- Logical propositions correspond to types.
- Proofs of propositions correspond to terms of the corresponding types.
- Composition of proofs via structural rules, including cut, corresponds to functional composition or substitution of terms.

This correspondence allows us to interpret type-theoretic derivations as constructive proofs and, conversely, to view logical proofs as typed programs. In the following sections, we will make this connection explicit by demonstrating how each type constructor ($\to$, $\times$, $+$, etc.) corresponds to a logical connective, and how typing rules correspond to natural deduction rules for the associated logic.

Moreover, this correspondence is more

For instance, consider the term

$$\lambda x.\lambda y.x : A \to B \to A$$

Via the correspondence

4 Inductive Types

foo

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 suffices 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:

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

- Computation:

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

  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.

Here's an example of using

4.1 General 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 4.1 (Constructor Discrimination)**.** *If $i \neq j$, then for all argument tuples $a_1,\dots,a_{n_i}$ and $b_1,\dots,b_{n_j}$:*

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

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

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

and

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

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

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

If we had $c_i(a_1,\dots,a_{n_i}) \equiv c_j(b_1,\dots,b_{n_j})$, 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 of inductive constructors, we first need a lemma showing that the sum constructors in particular are injective.

Lemma 4.2. *For any $a,x : A$ and $b,y : B$,*

$$\mathsf{inl}(a_1) \equiv \mathsf{inl}(a_2) \implies a \equiv x$$
$$\mathsf{inr}(b_1) \equiv \mathsf{inr}(b_2) \implies b \equiv y$$

Proof. We prove this for inl; the other case is analogous. If A is empty, then this is vacuous, so assume otherwise that A is inhabited and fix an arbitrary $a_0 : A$. Define the step functions

$$f_1 :\equiv \lambda x.x$$
$$f_2 :\equiv \lambda y.a_0$$

By the computation rules,

$$\mathsf{rec}^A_{A+B}(f_1,f_2)\big(\mathsf{inl}(a_1)\big) \equiv f_1(a_1) \equiv a_1$$
$$\mathsf{rec}^A_{A+B}(f_1,f_2)\big(\mathsf{inl}(a_2)\big) \equiv f_1(a_2) \equiv a_2$$

but by assumption, $\mathsf{inl}(a_1) \equiv \mathsf{inl}(a_2)$, so by congruence of equality under function application,

$$\mathsf{rec}^A_{A+B}(f_1,f_2)\big(\mathsf{inl}(a_1)\big) \equiv \mathsf{rec}^A_{A+B}(f_1,f_2)\big(\mathsf{inl}(a_2)\big)$$

and hence $a_1 \equiv a_2$. ■

Injectivity of constructors of larger sums is then obtained by induction on this result.

Theorem 4.3 (Injectivity of Constructors)**.** *If c_i is a constructor of T, then:*

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

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

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:

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

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, $A_{i,j}[B/T] \equiv B$,

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

  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.

This 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*.

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:

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

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

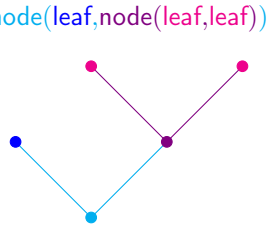
$$\mathsf{map}(g,\mathsf{nil}_A) \equiv \mathsf{nil}_B : \mathsf{List}(B)$$
$$\mathsf{map}(g,\mathsf{cons}_A(a,\ell)) \equiv \mathsf{cons}_B\big(g(a),\mathsf{map}(g,\ell)\big).$$

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.

Here is another inductive type, where the type appears nested more deeply than top-level in one of the constructors

$$\mathsf{leaf} : T \qquad \text{and} \qquad \mathsf{node} : T \times T \to T$$

Intuitively, a term of this type is either a leaf, or a node containing two subterms of the type; T is the type ofi *finite* (*unlabelled*) *binary trees*:



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 $f_2 : B \to C$. △

4.1.1 Polarity

Why is the function type not an inductive type?

The introduction rule for functions is somewhat unusual, in that it modifies the context, unlike the corresponding rules for products or sums. We obtain a term of type $A \to B$ by constructing, under the assumption of a term $x : A$, a term $b : B$, and in the resulting judgement, the assumption $x : A$ is *discharged*. The context thus mirrors the structure of *conditional reasoning*: assuming that an entity of type A exists, we establish that one of type B can be obtained; having done so, we may then speak of a general transformation from A to B.

This is because the construction of a function term involves a form of *abstraction*, rather than the simple composition of pre-existing data

We briefly mentioned in the previous section that there are some restrictions on the ways that the inductive type being defined can occur in its own constructors. This is because certain patterns of self-reference can lead to non-terminating or inconsistent definitions.

In particular, the type being defined cannot appear directly in the return type of a function.

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

$$t_0 : T \qquad \text{and} \qquad \mathsf{foo} : (A \to T) \to T$$

What happens when we try define the recursor to eliminate a term of this type? The recursor would have type

$$\mathsf{rec}_T^B : B \to \big((A \to B) \to B\big) \to T \to B$$

with computation rules of the form:

$$\mathsf{rec}_T^B(b_0,f)(t_0) \equiv b_0$$
$$\mathsf{rec}_T^B(b_0,f)\big(\mathsf{foo}(g)\big) \equiv f(g')$$

The problem is that, without restriction, these inductive types are consistent with the existence of non-well-founded terms. For example,

where $g' : A \to B$ should be somehow obtained by pushing the recursion through the structure of $g : A \to T$.

When constructing an inductive type with a constructor like

$$\mathsf{bar} : A \times T \to T$$

and build up a value like $\mathsf{bar}(a,t)$, we must explicitly *supply* concrete arguments $a : A$ and $t : T$. In contrast, with a constructor like

$$\mathsf{foo} : (A \to T) \to T$$

if we try build up a value $\mathsf{foo}(g)$,

However, the inductive definition of T only tells us what to do with *values* of type T; we don't know how to break down a *function* $g : \mathsf{unit} \to T$ into simpler pieces to start this structural recursion.

to supply such an argument requires that we already know how to produce elements of T; we are trying to define T in terms of a function that already requires T to exist to build its argument.

However, consider what the lift_T operation would require for functions:

$$\mathsf{lift}_{X_1 \to X_2}(h) : (X_1 \to X_2) \to (X_1' \to X_2')$$

$$\mathsf{lift}_{X_1 \to X_2}(h)(f) :\equiv \lambda x'.\mathsf{lift}$$

Even if we tried manually

Since we have no strictly

Suppose we have

$$g(x) \equiv \mathsf{foo}(g) : T$$

What if we try to apply the recursor to this term?

$$\mathsf{rec}_T^B(b_0, f)\big(\mathsf{foo}(g)\big) \equiv$$

If we try to define

$$\mathsf{rec}_T^B(b_0, f)(\mathsf{foo}(g)) \equiv f(\mathsf{rec}_T^B(b_0, f)(g))$$

then

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

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

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

4.2 Derivations

5 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.

5.1 Dependent Sums

5.2 Dependent Products

5.3 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

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

5.4 Quantifiers

5.5 Girard's Paradox

Addendum

ree