

URSS

RESEARCH OUTPUT

Introduction to Type Theory

Type Theory for Mathematics

2025-10-09

Rin Liu

Contents

Table of Contents	ii
Foreword	iv
1 Introduction	1
1.1 Typing as Annotation	2
1.2 Intrinsic and Extrinsic Typing	3
1.3 Typing as Specification	4
2 Simple Type Theory	4
2.1 Judgements	5
2.1.1 Types and Contexts	6
2.1.2 Sequents	7
2.1.3 Judgemental Equality	7
2.2 Types	9
2.2.1 Function Types	9
2.2.2 Product Types	10
2.2.3 Sum Types	10
2.2.4 More Base Types	11
2.3 Derivations	12
2.4 Inversion	13
2.5 Admissible Rules	14
3 Metatheory	16
3.1 Operational Semantics	16
3.1.1 Normal Forms	18
3.1.2 Strong Normalisation	20
3.2 Confluence	25
3.3 Unification	32
4 The Curry-Howard Correspondence	35
4.1 Term Erasure	36
4.2 Cut and β -Reduction	40
4.3 Program = Proof	42
5 Inductive Types	44
5.1 nat Revisited	44
5.2 General Inductive Types	44
5.2.1 Polarity	53
5.3 Derivations	55
6 Dependent Type Theory	56
6.1 Judgements	56
6.2 Universe Types	56
6.3 Dependent Sums	56
6.4 Dependent Products	56
6.5 Inductive Types	56
6.6 Inductive-Inductive Types	56
6.7 Quantifiers	56
6.8 Girard's Paradox	56
Addendum	57

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 $\text{prime}(x)$ 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:

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

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

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

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

$$\in : \mathcal{U} \times \mathcal{U} \rightarrow \mathbf{Prop}$$

where \mathcal{U} denotes the universe of sets and \mathbf{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 $+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, 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 $\{\emptyset, \{\emptyset\}\}$ is simultaneously:

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

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

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

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:

$$\langle a, b \rangle := \{\{a\}, \{a, b\}\}.$$

and the corresponding projections can be defined as:

$$\begin{aligned}\pi_1 \langle a, b \rangle &= \bigcap \langle a, b \rangle \\ \pi_2 \langle a, b \rangle &= \bigcup \langle a, b \rangle \setminus \bigcap \langle a, b \rangle\end{aligned}$$

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 :

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

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:

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

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 $f : A \rightarrow B$ 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.

2.1.1 Types and Contexts

Just as terms are introduced and reasoned about through typing judgements, the formation of types themselves are also controlled by judgements. 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 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, \dots that always satisfy this judgement without any assumptions. These primitive types are given only *opaquely*: we know they exist, and we can declare terms to inhabit them in assumptions, but their internal structure is unspecified. Later, we will introduce more complex types that will be formed from these base types, as well as other non-opaque base types. For now, the theory treats each of these types as an indivisible symbol whose sole role is to serve as a target for typing declarations.

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

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

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

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

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

Γ ctx

This judgement is governed by the following inference rule:

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

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.

2.1.2 Sequents

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, and not *existential* claims. 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

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

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.

2.1.3 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 $f := \lambda x.x + 1$ being applied to an argument 2, then the expressions

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

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:

$$n + 0 \equiv n, \quad n + \text{succ}(m) \equiv \text{succ}(n + m)$$

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

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

In each case, the equality is *syntactic*, and is not something that requires constructing any kind of proof term. Intuitively, judgemental equality captures all the “obvious” kinds of equalities that can be obtained by unfolding definitions or mechanically applying computation rules. We will focus more on this computational interpretation of equality in a later section on metatheory.

There is also a notion of judgemental equality for types

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

and also for contexts:

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

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:

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

and analogous rules hold for types and contexts, replacing “ $- : A$ ” with “ $- \text{ type}$ ” and “ $- \text{ ctx}$ ”, respectively.

Moreover, all three are *congruence* relations; judgementally equal terms may be substituted in any syntactic construction within their own sorts. For instance, if $a \equiv b : A$ and $f : A \rightarrow B$, then $f(a) \equiv f(b) : B$.

All three forms also interact with *each other* coherently: each is stable under substitution of judgementally equal objects of the other sorts. For instance, if $\Gamma \vdash a \equiv b : A$ and $\Gamma \equiv \Delta \text{ ctx}$, then $\Delta \vdash a \equiv b : A$:

$$\frac{\Gamma \vdash a \equiv b : A \quad \Gamma \equiv \Delta \text{ ctx}}{\Delta \vdash a \equiv b : A}$$

and similar for all other possible pairings.

Finally, all three forms of equality judgement are stable under α -conversion – that is, renaming of bound variables does not affect the validity of any equality judgement.

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

Formally, in terms of judgements:

$$\frac{A \text{ type} \quad B \text{ type}}{A \rightarrow B \text{ type}} \quad \frac{A \text{ type} \quad B \text{ type}}{A \times B \text{ type}} \quad \frac{A \text{ type} \quad B \text{ type}}{A + B \text{ type}}$$

We now give the rules for each compound type.

2.2.1 Function Types

- Introduction:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B} (\rightarrow_I)$$

Given a term $t : B$ depending on free variable $x : A$, we may form a term of type $A \rightarrow B$ by *abstracting* over x .

- Elimination:

$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f a : B} (\rightarrow_E)$$

Given a function $f : A \rightarrow B$, 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:

$$\overline{\Gamma \vdash (\lambda x. t) a \equiv t[a/x]} (\rightarrow_\beta)$$

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:

$$\overline{\Gamma \vdash f \equiv \lambda x. f(x)} (\rightarrow_\eta)$$

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 $A \rightarrow B$ 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 $\lambda x. t$ to obtain a function of type $A \rightarrow B$. 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 $A \times B$.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B} (\times_1)$$

- Elimination: from a term of a product type, we can obtain terms of each component type by projection:

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \text{fst}(p) : A} (\times_{E_1}) \quad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \text{snd}(p) : B} (\times_{E_2})$$

- Computation:

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

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:

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

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:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{inl}(a) : A + B} (+_{I_1}) \quad \frac{\Gamma \vdash b : B}{\Gamma \vdash \text{inr}(b) : A + B} (+_{I_2})$$

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

- Destructors:

$$\frac{\Gamma \vdash t : A + B \quad \Gamma \vdash f : A \rightarrow C \quad \Gamma \vdash g : B \rightarrow C}{\Gamma \vdash \text{case}(t; f, g) : C} (+_E)$$

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:

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

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

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

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 $\star : \text{unit}$, or in computer science contexts, by $() : \text{unit}$. 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:

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

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

- Elimination: to define a function $\text{unit} \rightarrow B$, it suffices to provide a canonical representative $b : B$:

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

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

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

- Uniqueness: since `unit` only has one constant constructor, every term of type `unit` is definitionally equal 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:

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

The destructor $\text{abort} : \text{void} \rightarrow B$ 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:

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

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 $\lambda x. \langle x, x \rangle : A \rightarrow A \times A$ is derivable:

$$\frac{\frac{\frac{}{x : A \vdash x : A} \text{ (axiom)} \quad \frac{}{x : A \vdash x : A} \text{ (axiom)}}{x : A \vdash \langle x, x \rangle : A \times A} (\times_I)}{\lambda x. \langle x, x \rangle : A \rightarrow A \times A} (\rightarrow_I)$$

△

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

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

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.

2.4 Inversion

The introduction and elimination rules we have seen so far are all *syntactically directed*: the form of a well-typed term uniquely determines the rule by which it must have been introduced, and hence determines certain structural facts about its type. That is, there are no notational collisions between any of these rules. This allows us to “invert” some inference rules and deduce the premises from the conclusion.

For instance, a term of the form $\lambda x.u$ necessarily must have arisen via an introduction rule for function types, and therefore its type must be a function type. Similarly, if a term is a pair, an application, or a case distinction, its typing can only have arisen from the corresponding introduction or elimination rule, and thus one can deduce the types of its subterms and the relevant constraints on the typing context.

This also applies recursively into subterms of the type. For instance, if we were given the term $\lambda x.\langle x, x \rangle : T$ from the previous example, we can deduce that T must be a function type $A \rightarrow B$ with $x : A$; and given the form of the function body $\langle x, x \rangle$ and typing $x : A$, we can deduce that $B \equiv A \times A$; so we must have $T \equiv A \rightarrow A \times A$.

This idea that the syntactic form of a term determines the possible forms of its type T is formalised by the *inversion lemma*:

Lemma 2.1 (Inversion). *Given $\Gamma \vdash t : T$, the form of T may be deduced from the syntactic form of t . Specifically:*

- if $\Gamma \vdash x : T$ and x is a variable, then $x : T$ is declared in Γ ;
- if $\Gamma \vdash \lambda x.u : T$, then there exist types A and B such that $T \equiv A \rightarrow B$ and $\Gamma, x : A \vdash u : B$;
- if $\Gamma \vdash uv : T$, then there exists a type A such that $\Gamma \vdash u : A \rightarrow T$ and $\Gamma \vdash v : A$;
- if $\Gamma \vdash \langle a, b \rangle : T$, then there exist types A and B such that $T \equiv A \times B$, and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$;
- if $\Gamma \vdash \text{fst}(p) : T$ or $\Gamma \vdash \text{snd}(p) : T$, then there exists A such that $\Gamma \vdash p : T \times A$ or $\Gamma \vdash p : A \times T$, respectively;
- if $\Gamma \vdash \text{inl}(a) : T$ or $\Gamma \vdash \text{inr}(b) : T$, then there exist types A and B such that $T \equiv A + B$, and $\Gamma \vdash a : A$ or $\Gamma \vdash b : B$, respectively;
- if $\Gamma \vdash \text{case}(t; f, g) : T$, then there exist types A and B such that $\Gamma \vdash t : A + B$, $\Gamma \vdash f : A \rightarrow T$, and $\Gamma \vdash g : B \rightarrow T$.

Proof. By induction on the typing derivation of $\Gamma \vdash t : T$. We only give the proof for abstractions.

- $\Gamma \vdash x : T$ with x a variable. The only rule that produces a judgement with a variable at the head is the axiom rule, so $x : T$ is necessarily declared in Γ .
- $\Gamma \vdash \lambda x.u : T$. The only rule that produces an λ -abstraction at the top level is \rightarrow_I , whose conclusion is a judgement of the form $\Gamma \vdash \lambda x.u : A \rightarrow B$, so necessarily T is equal to some $A \rightarrow B$ with $\Gamma, x : A \vdash u : B$.
- All other cases proceed similarly: since the introduction and elimination rules of our theory are all syntax-directed, only one rule can introduce a given syntactic form at the head of a typing derivation, and hence each case is reduced to a unique inversion determined by the last typing rule applied.

■

The inversion lemma ensures that typing judgements are well behaved with respect to the syntactic structure of terms: every typing derivation is sufficiently constrained by the form of the term being typed. This property will be useful in later metatheoretic results, as it allows us to reason backwards from judgements to their possible derivations.

2.5 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 $\Gamma, y : B \vdash x : A$. More generally, we can insert the extra assumption anywhere in the context:

Theorem 2.2 (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

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

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

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

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

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

$$\frac{\Gamma, z : C \vdash t : D}{\Gamma \vdash \lambda z. t : C \rightarrow D} (\rightarrow_I)$$

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

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

and abstraction again yields

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

as required.

- If the last rule is application:

$$\frac{\Gamma \vdash t_1 : C \rightarrow A \quad \Gamma \vdash t_2 : C}{\Gamma \vdash t_1 t_2 : A} (\rightarrow_E)$$

then by the induction hypothesis, we have:

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

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:

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

- 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:

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

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.

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

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 more general list-based representation is the standard definition, with the set-like behaviour of contexts in our particular simple type theory being recovered as a theorem.

Theorem 2.3. *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*.

3 Metatheory

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 all of the proofs in this section on first reading. All subsequent sections can be followed without reference to these exact metatheoretic details; the proofs 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.

In this section only, we will use the equality symbol $=$ to denote exact syntactic equality, up to α -conversion.

3.1 Operational Semantics

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, a notion of equality can be defined *operationally* in terms of this reduction system, where two terms are computationally equal or “ (β) -convertible” if they compute to a common term.

In a well-behaved rewriting system – one in which reduction is *normalising* and *confluent* – these two notions of equality coincide: every equality derivable from the primitive inference rules is witnessed by reduction to a common form, and conversely, every pair of terms that reduce to a common form is derivably judgementally 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:

$$\begin{array}{ll}
 (\lambda x.t)u \longrightarrow t[u/x] & (\rightarrow_\beta) \\
 \text{fst}\langle a,b \rangle \longrightarrow a & (\times_{\beta_1}) \\
 \text{snd}\langle a,b \rangle \longrightarrow b & (\times_{\beta_2}) \\
 \text{case}(\text{inl}(a),f,g) \longrightarrow f\ a & (+_{\beta_1}) \\
 \text{case}(\text{inr}(b),f,g) \longrightarrow g\ b & (+_{\beta_2})
 \end{array}$$

and under substitution (“*congruence of reduction*”):

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

where $C[t]$ is a *term context* denoting 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,

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

Reductions also preserve well-typedness, and in particular, typing annotations:

Lemma 3.1 (Subject Reduction). *If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.*

Proof. By structural induction on the derivation of $t \longrightarrow t'$.

For the β rules:

- (\rightarrow_β) $\Gamma \vdash (\lambda x.t)u : B$ and $(\lambda x.t)u \longrightarrow t[u/x]$. By inversion on application, there exists a type A with $\Gamma \vdash \lambda x.t : A \rightarrow B$ and $\Gamma \vdash u : A$, and by inversion on abstraction, we obtain $\Gamma, x : A \vdash t : B$:

$$\frac{\frac{\vdots}{\Gamma, x : A \vdash t : B}}{\Gamma \vdash \lambda x.t : A \rightarrow B} \quad \frac{\vdots}{\Gamma \vdash u : A} \\
 \Gamma \vdash (\lambda x.t)u : B$$

so by the cut rule applied to $\Gamma \vdash u : A$ and $\Gamma, x : A \vdash t : B$, we obtain $\Gamma \vdash t[u/x] : B$.

- (\times_β) $\Gamma \vdash \text{fst}\langle a,b \rangle : A$ and $\text{fst}\langle a,b \rangle \longrightarrow a$. The proof for snd is analogous.

By inversion on projections, there exists a type B such that $\Gamma \vdash \langle a,b \rangle : A \times B$, so by inversion on pairing, $\Gamma \vdash a : A$.

- $(+_{\beta}) \Gamma \vdash \text{case}(\text{inl}(a), \lambda x.f, \lambda x.g) : C$ and $\text{case}(\text{inl}(a), \lambda x.f_0, g) \longrightarrow f_0[a/x]$ (for simplicity, we are binding x in f explicitly; this assumption is justified by the next lemma on canonical forms). The proof for $\text{inr}(b)$ is analogous.

By inversion on case analysis, $\Gamma \vdash \text{inl}(a) : A + B$, $\Gamma \vdash \lambda x.f_0 : A \rightarrow C$, and $\Gamma \vdash g : B \rightarrow C$. By inversion on the left sum injection, $\Gamma \vdash a : A$, and by inversion on $\lambda x.f_0$, we have $\Gamma, x : A \vdash f_0 : C$. So by cut, $\Gamma \vdash f[a/x] : C$.

For congruence: $\Gamma \vdash C[t] : T$ and $t \longrightarrow t'$. We induct on the structure of the term context $C[-]$.

- Reduction in the function position: $C[-] = [-]u$ with $\Gamma \vdash tu : B$. By inversion on application, there exists a type A such that $\Gamma \vdash t : A \rightarrow B$ and $\Gamma \vdash u : B$. By the inductive hypothesis, $\Gamma \vdash t' : A \rightarrow B$, so by \rightarrow_E , $\Gamma \vdash t'u : B$.
- Reduction in the argument position: $C[-] = f[-]$ with $\Gamma \vdash ft : B$. By inversion, $\Gamma \vdash f : A \rightarrow B$ and $\Gamma \vdash t : A$. By the inductive hypothesis, $\Gamma \vdash t' : A$, so $\Gamma \vdash ft' : B$.
- All other term context forms (reduction inside pairs, projections, etc.) are handled by the same pattern: apply inversion to obtain typings for the immediate subterms, apply the induction hypothesis to the reduced subterm, then reassemble the typing for the whole term using the appropriate typing rule.

■

This proof proceeds by inversion on the typing derivations for each reduction form. In our current formulation, the computational equalities corresponding to β -rules are built directly into the definition of judgemental equality, so subject reduction for these reductions follows immediately and the inversion arguments for direct computation steps are technically unnecessary, with only the final induction on term contexts being required.

Nevertheless, this longer inversion-based proof is valuable because it applies to more general type theories in which computation is not built into the typing rules. More importantly, this approach is robust: the simpler proof that relies on β -rules will not extend to the more expressive type theories we will explore later, whereas the inversion-based method will still hold.

3.1.1 Normal Forms

A term t is 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”.

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.

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

While the normal form of terms of base types is straightforward, it is not immediately obvious what the normal form of a term of a compound type should look like. Forming compound terms like applications and case analyses may produce redexes if their subterms are not in canonical form, so it is not straightforward to determine a general structure for these normal forms.

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, since they may contain free variables, causing “stuck” unevaluated subterms.

However, it turns out that every *closed* normal form is in fact canonical.

Lemma 3.2 (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 $T \equiv A \rightarrow B$, then $t = \lambda x.u$ for some $x : A$ and $u : B$;
- (iii) If $T \equiv A \times B$, then $t = \langle a, b \rangle$ for some $a : A$ and $b : B$;
- (iv) If $T \equiv A + B$, then either $t = \text{inl}(a)$ for some $a : A$, or $t = \text{inr}(b)$ 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. $\star : \text{unit}$);
- 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 $x : A \rightarrow B$;
- (\rightarrow_I) a λ -abstraction $\lambda x.u$;
- (\rightarrow_E) an application $u v$ where $u : C \rightarrow (A \rightarrow B)$ and $v : C$.
- (\times_E) a projection $\text{fst}(x)$ with $x : (A \rightarrow B) \times C$, or $\text{snd}(y)$ with $y : C \times (A \rightarrow B)$;
- ($+_E$) a case analysis $\text{case}(x; f, g)$ with $x : C + D$, $f : C \rightarrow A \rightarrow B$, and $g : D \rightarrow A \rightarrow B$.

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 $t = \lambda x.u$. 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 $t = \lambda x.u$ with canonical subterms.

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

- (axiom) a variable $x : A \times B$;
- (\times_I) a pair $\langle a, b \rangle$;
- (\times_E) a projection $\text{fst}(x)$ with $x : (A \times B) \times C$, or $\text{snd}(y)$ with $y : C \times (A \times B)$;
- (\rightarrow_E) an application $u v$ where $u : C \rightarrow (A \times B)$ and $v : C$.
- ($+_E$) a case analysis $\text{case}(x; f, g)$ with $x : C + D$, $f : C \rightarrow (A \times B)$, and $g : D \rightarrow (A \times B)$.

The first and last three cases are excluded as above, so we must have $t = \langle a, b \rangle$. 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 $x : A + B$;
- ($+_I$) an injection $\text{inl}(a)$ with $a : A$, or $\text{inr}(b)$ with $b : B$;
- (\rightarrow_E) an application $u v$ where $u : C \rightarrow (A + B)$ and $v : C$.
- (\times_E) a projection $\text{fst}(x)$ with $x : (A + B) \times C$, or $\text{snd}(y)$ with $y : C \times (A + B)$;

- $(+_E)$ a case analysis $\text{case}(x; f, g)$ with $x : C + D$, $f : C \rightarrow (A + B)$, and $g : D \rightarrow (A + B)$.

The first and last three cases are excluded as above, so we must have $t = \text{inl}(a)$ or $t = \text{inr}(b)$. 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 $\text{inl}(a)$ or $\text{inr}(b)$, 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 $A \rightarrow B$ 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.

3.1.2 Strong Normalisation

The *multi-step reduction* relation $\xrightarrow{*}$ is the reflexive transitive closure of \rightarrow . That is, the smallest relation such that:

- if $t \rightarrow t'$, then $t \xrightarrow{*} t'$;
- for all terms t , $t \xrightarrow{*} t$;
- and if $t \xrightarrow{*} t'$ and $t' \xrightarrow{*} t''$, then $t \xrightarrow{*} t''$.

The *reductive equality* or β -*convertibility* relation \equiv_β is the reflexive, symmetric, transitive closure of \rightarrow , or equivalently, the symmetric closure of $\xrightarrow{*}$. That is, the smallest relation such that:

- if $t \xrightarrow{*} t'$, then $t \equiv_\beta t'$ and $t' \equiv_\beta t$.

Intuitively, two terms are β -equivalent if there is a reduction path from one to the other, where we allow the use of reduction rules in reverse:

$$t \xrightarrow{*} t_1 \xleftarrow{*} t_2 \xrightarrow{*} t_3 \xleftarrow{*} \cdots \xrightarrow{*} t_{n-1} \xleftarrow{*} u$$

By transitivity of $\xrightarrow{*}$, we may assume that these reductions reverse direction between every intermediate term in any such sequence.

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

$$t \xrightarrow{*} t'$$

Example. The term

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

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

$$\Omega = (\lambda x. \textcolor{blue}{x} \textcolor{red}{x})(\lambda x. x x) \rightarrow (\lambda \textcolor{blue}{x}. \textcolor{blue}{x} \textcolor{red}{x})(\lambda \textcolor{blue}{x}. \textcolor{red}{x} x) = \Omega$$

Conversely, the term

$$(\lambda x. y) \Omega$$

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

$$(\lambda x.y) \Omega \longrightarrow y$$

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

$$(\lambda x.y) \Omega \longrightarrow (\lambda x.y) \Omega \longrightarrow (\lambda x.y) \Omega \longrightarrow \dots$$

△

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. This strategy is often efficient in practice because it avoids repeating work and never generates any additional thunks. However, it also always evaluates all function arguments, which can lead to wasted computation when arguments are unused.
- *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 $(\lambda x.y) \Omega$.

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.

The reason that normal-order reduction always finds a normal form if one exists 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.

However, rigorously establishing that normal-order reduction always reaches a normal form for weakly normalising terms is technically involved and relies on detailed syntactic arguments about the order

of redex contractions. Reasoning along these lines quickly becomes cumbersome in developing later metatheoretic results; it would be preferable to avoid dependence on any specific reduction strategy altogether.

Because of this, weak normalisation is too fragile of a notion to support deeper metatheoretic reasoning, since the mere existence of a normal form does not guarantee that it can be reached by an arbitrary reduction strategy.

For our purposes, 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:

$$t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \dots$$

Equivalently, every reduction sequence terminates in finitely many steps.

Lemma 3.3. *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 $t \longrightarrow t'$ 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 substitute Ω for x . △

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

A term t in context $\Gamma \equiv x_i : A_i, \dots, x_n : A_n$ 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 $\text{SN}(t)$ to denote that a closed term t is strongly normalising, while for an open term t in context Γ , we write $\text{SN}_\Gamma(t)$ to denote that t is strongly normalising under all strongly normalising closed substitutions in Γ .

So far, we have discussed terms in our rewriting system as isolated syntactic constructions. What happens if we add types back in?

Theorem 3.4 (Strong Normalisation). *Every well-typed term is strongly normalising. That is, if $\Gamma \vdash t : A$, then $\text{SN}_\Gamma(t)$.*

It turns out that for the typed fragment of this rewriting system, *every* term is strongly normalising; every computation will eventually terminate, regardless of the choice of evaluation strategy (though different strategies may take different amounts of time and memory to terminate).

The proof strategy for this result is standard in the literature, 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 $\llbracket A \rrbracket$ of strongly normalising closed terms of type A , called the *reducibility candidate* of A . We call elements of these sets *reducible* terms. These sets are defined via structural recursion on A as follows:

- Base types A :

$$\llbracket A \rrbracket := \{t : A \mid \text{SN}(t)\}$$

as all basic terms cannot reduce further;

- Function types $A \rightarrow B$:

$$\llbracket A \rightarrow B \rrbracket := \{f : A \rightarrow B \mid \text{SN}(f) \wedge \forall a \in \llbracket A \rrbracket. f a \in \llbracket B \rrbracket\}$$

so that t behaves well on all reducible arguments:

- Product types $A \times B$:

$$\llbracket A \times B \rrbracket := \{t : A \times B \mid \text{SN}(t) \wedge \text{fst}(t) \in \llbracket A \rrbracket \wedge \text{snd}(t) \in \llbracket B \rrbracket\}$$

so that components are reducible;

- Sum types $A + B$:

$$\llbracket A + B \rrbracket := \{t : A + B \mid \text{SN}(t) \wedge \forall C \text{ type}. \forall f \in \llbracket A \rightarrow C \rrbracket. \forall g \in \llbracket B \rightarrow C \rrbracket. \text{case}(t; f, g) \in \llbracket C \rrbracket\}$$

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 3.5 (Closure under Reduction). *If $t \in \llbracket A \rrbracket$ and $t \longrightarrow t'$, then $t' \in \llbracket A \rrbracket$.*

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 $A \rightarrow B$: for all $a \in \llbracket A \rrbracket$, we have by congruence of reduction:

$$t a \rightarrow t' a$$

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

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

$$\text{fst}(t) \longrightarrow \text{fst}(t'), \quad \text{snd}(t) \longrightarrow \text{snd}(t')$$

By the inductive hypothesis on A and B , $\text{fst}(t') \in \llbracket A \rrbracket$ and $\text{snd}(t') \in \llbracket B \rrbracket$, and hence $t' \in \llbracket A \times B \rrbracket$.

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

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

By the inductive hypothesis on C , we have $\text{case}(t'; f, g) \in \llbracket C \rrbracket$ and hence $t' \in \llbracket A + B \rrbracket$

■

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 escape 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 3.6 (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 $\sigma(x_i) \in \llbracket A_i \rrbracket$, then the closed term $t[\sigma]$ belongs to $\llbracket A \rrbracket$.*

Proof. By induction on the typing derivation of t :

- Axiom $t = x_i : A_i$. By assumption, $t[\sigma] = \sigma(x_i) \in \llbracket A_i \rrbracket$.
- $(\rightarrow_I) t = \lambda x. u : A \rightarrow B$. Let $a \in \llbracket A \rrbracket$. Then, $(\lambda x. u)[\sigma] a \rightarrow u[\sigma, a/x]$. 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 $u[\sigma, a/x] \in \llbracket B \rrbracket$. As $a \in \llbracket A \rrbracket$ was arbitrary, we have $(\lambda x. u)[\sigma] = t[\sigma] \in \llbracket A \rightarrow B \rrbracket$.
- $(\rightarrow_E) t = u v : B$. By the inductive hypothesis on the premises $\Gamma \vdash u : A \rightarrow B$ and $\Gamma \vdash v : A$, we have $u[\sigma] \in \llbracket A \rightarrow B \rrbracket$ and $v[\sigma] \in \llbracket A \rrbracket$. By the function clause, $u[\sigma] v[\sigma] = t[\sigma] \in \llbracket B \rrbracket$.
- $(\times_I) t = \langle u, v \rangle : A \times B$. By the inductive hypothesis on $\Gamma \vdash u : A$ and $\Gamma \vdash v : B$, we have $u[\sigma] \in \llbracket A \rrbracket$ and $v[\sigma] \in \llbracket B \rrbracket$. Since the components are strongly normalising and the pair constructor doesn't introduce infinite reductions, $t[\sigma] = \langle u, v \rangle[\sigma] = \langle u[\sigma], v[\sigma] \rangle$ is strongly normalising.

Since reducibility candidates are closed under reduction, \times_β yields $\text{fst}\langle u[\sigma], v[\sigma] \rangle = u[\sigma] \in \llbracket A \rrbracket$ and $\text{snd}\langle u[\sigma], v[\sigma] \rangle = v[\sigma] \in \llbracket B \rrbracket$, so $\langle u[\sigma], v[\sigma] \rangle = \langle u, v \rangle[\sigma] = t[\sigma] \in \llbracket A \times B \rrbracket$.

- $(\times_E) t = \text{fst}(u) : A$ (the proof for $t = \text{snd}(v) : B$ is analogous). By the inductive hypothesis on $\Gamma \vdash u : A \times B$, we have $u[\sigma] \in \llbracket A \rrbracket$, so $\text{fst}(u[\sigma]) = \text{fst}(u)[\sigma] = t[\sigma] \in \llbracket A \rrbracket$.
- $(+_I) t = \text{inl}(u) : A + B$ (the proof for $t = \text{inr}(v)$ is analogous). By the inductive hypothesis on $\Gamma \vdash u : A$, we have $u[\sigma] \in \llbracket A \rrbracket$, so $\text{inl}(u[\sigma]) = t[\sigma]$ is strongly normalising.

Let $f \in \llbracket A \rightarrow C \rrbracket$ and $g \in \llbracket B \rightarrow C \rrbracket$. Then, $\text{case}(\text{inl}(u[\sigma]); f, g) \in \llbracket C \rrbracket$. By $+\beta$, $\text{case}(\text{inl}(u); f, g) = f(u[\sigma])$, and since $f \in \llbracket A \rightarrow C \rrbracket$ and $u[\sigma] \in \llbracket A \rrbracket$, we have $f(u[\sigma]) \in \llbracket C \rrbracket$. As $f \in \llbracket A \rightarrow C \rrbracket$ and $g \in \llbracket B \rightarrow C \rrbracket$ were arbitrary, we have $\text{inl}(u)[\sigma] = t[\sigma] \in \llbracket A + B \rrbracket$.

- $(+_E) t = \text{case}(u; \lambda x. v, \lambda y. w) : C$. By the inductive hypothesis on the three premises $\Gamma \vdash u : A + B$, $\Gamma \vdash f : A \rightarrow C$, and $\Gamma \vdash g : B \rightarrow C$, we have $u[\sigma] \in \llbracket A + B \rrbracket$, $f[\sigma] \in \llbracket A \rightarrow C \rrbracket$, and $g[\sigma] \in \llbracket B \rightarrow C \rrbracket$, respectively.

By the sum clause, $\text{case}(u[\sigma]; f[\sigma], g[\sigma]) = \text{case}(u; f, g)[\sigma] = t[\sigma] \in \llbracket C \rrbracket$.

■

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 candidate when instantiated with any reducible substitution. This observation provides the key link needed to establish strong normalisation for all well-typed terms.

Theorem 3.4 (Strong Normalisation). *Every well-typed term is strongly normalising. That is, if $\Gamma \vdash t : A$, then $\text{SN}_\Gamma(t)$.*

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

This gives us a full guarantee that infinite loops and non-terminating evaluations cannot occur in our theory – in particular, any function definable in this theory is necessarily total, as every term representing a computation is guaranteed to produce a result.

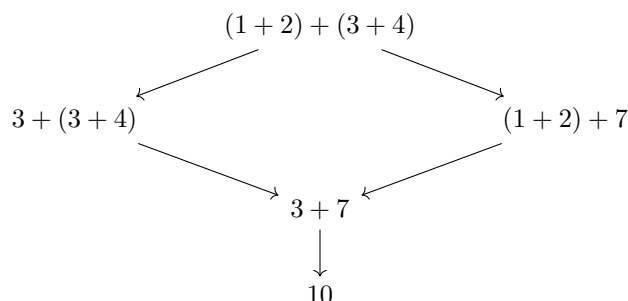
This also implies that non-normalising terms like the Ω combinator are not typable. In untyped λ calculus, Ω is the canonical example of a term with no normal form by exhibiting infinite self-application – but such a term is prohibited at the syntactic level by types.

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.

3.2 Confluence

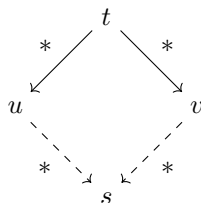
While strong normalisation ensures that every reduction sequence terminates, it does not by itself guarantee that this process is *deterministic*: it may be the case that distinct reduction paths from a common term could converge to distinct normal forms. In such a situation, although every computation eventually halts, its outcome may depend on the order in which reductions are applied. For a rewriting system to be reasonably useful as a model of computation, we should be able to speak about “the” result of a reduction.

For instance, the result of computing $(1 + 2) + (3 + 4)$ is unambiguously 10, because it doesn’t matter how we reduce the expression:



Without any guarantee of determinism, this kind of ambiguity would undermine our notion of reductive equality: two distinct normal forms could be β -convertible to a common ancestor, yet not to each other by reduction. Consequently, normal forms would not serve as canonical representatives of equivalence classes under reduction, and the notion of computation would lose its intended meaning.

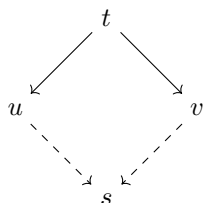
A rewriting system is *confluent* if, whenever a term t reduces to two terms u and v , then u and v also reduce to some common term s :



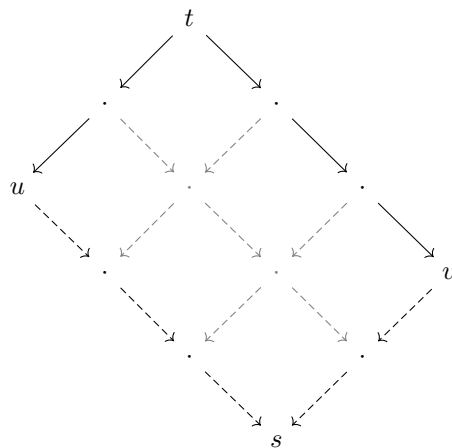
That is, from the same starting point, different reduction sequences may produce distinct intermediate terms, but there is always a way for those reduction sequences to converge back to a common term.

From a broader perspective, confluence guarantees the global coherence of reduction: it establishes that, although its operational semantics may allow for locally distinct reductions at any given point, all possible paths ultimately describe the same global computational behaviour. This property is particularly important for the model- and proof-theoretic aspects of type theory, since it ensures that normal forms are indeed canonical representatives of β -convertibility classes, and that reductive equality corresponds to a single, well-determined notion of computation.

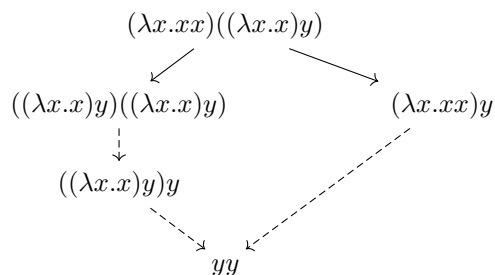
If the analogous *local* property for one-step reductions held:



then it would be easy to prove confluence, as we could just fill in the one-step “grid” of reductions spanned by the original reductions sequences to u and v :



Unfortunately, this is not the case. For instance, in the following reduction diagram, the left path requires at least two further reductions to join back up with the right path:



The problem here is that redexes can overlap: reducing one can destroy, duplicate, or otherwise alter others. Consequently, the effects of contracting distinct redexes are not independent, and the order in which reductions are applied can produce distinct intermediate terms that are not immediately reconcilable by a further single-step reduction, since any duplicated redexes need to be resolved separately. This lack of locality renders one-step reduction inherently non-commutative. To resolve this, we will instead consider a coarser notion of reduction.

The *parallel reduction* relation \longrightarrow is the smallest binary relation on terms such that:

$$\frac{}{x \longrightarrow x} \text{ (var}^{\parallel}\text{)}$$

$$\begin{array}{c}
\frac{t \longrightarrow t'}{\lambda x.t \longrightarrow \lambda x.t'} (\rightarrow_I) \quad \frac{t \longrightarrow t' \quad u \longrightarrow u'}{t u \longrightarrow t' u'} (\rightarrow_E) \\
\\
\frac{t \longrightarrow t' \quad u \longrightarrow u'}{(\lambda x.t)u \longrightarrow t'[u'/x]} (\rightarrow_\beta) \\
\\
\frac{t \longrightarrow t' \quad u \longrightarrow u'}{\langle t, u \rangle \longrightarrow \langle t', u' \rangle} (\times_I) \quad \frac{t \longrightarrow t'}{\text{fst}(t) \longrightarrow \text{fst}(t')} (\times_{E_1}) \quad \frac{t \longrightarrow t'}{\text{snd}(t) \longrightarrow \text{snd}(t')} (\times_{E_2}) \\
\\
\frac{t \longrightarrow t'}{\text{fst}\langle t, u \rangle \longrightarrow t'} (\times_{\beta_1}) \quad \frac{u \longrightarrow u'}{\text{snd}\langle t, u \rangle \longrightarrow u'} (\times_{\beta_2}) \\
\\
\frac{t \longrightarrow t'}{\text{inl}(t) \longrightarrow \text{inl}(t')} (+_{I_1}) \quad \frac{t \longrightarrow t'}{\text{inr}(t) \longrightarrow \text{inr}(t')} (+_{I_2}) \quad \frac{t \longrightarrow t' \quad f \longrightarrow f' \quad g \longrightarrow g'}{\text{case}(t; f, g) \longrightarrow \text{case}(t'; f', g')} (+_E) \\
\\
\frac{t \longrightarrow t' \quad f \longrightarrow f'}{\text{case}(\text{inl}(t); \lambda x.f, \lambda x.g) \longrightarrow f'[t'/x]} (+_{\beta_1}) \quad \frac{t \longrightarrow t' \quad g \longrightarrow g'}{\text{case}(\text{inl}(t); \lambda x.f, \lambda x.g) \longrightarrow g'[t'/x]} (+_{\beta_2})
\end{array}$$

That is, $t \longrightarrow u$ if u can be obtained by reducing any number of redexes in t *simultaneously*. Note that we are not obligated to reduce *all* redexes in t simultaneously; parallel reduction can reduce any subset of immediate redexes – including none:

Lemma 3.7 (Parallel Reflexivity). *For any term t , $t \longrightarrow t$.*

Proof. By induction on the derivation of t . ■

Parallel reduction is also compatible with substitution:

Lemma 3.8 (Parallel Substitution). *If $t \longrightarrow t'$ and $u \longrightarrow u'$, then $t[u/x] \longrightarrow t'[u'/x]$*

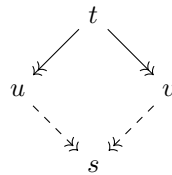
Proof. By induction on the derivation of $t \longrightarrow t'$. ■

The ordinary one-step reduction relation is too fine to satisfy this notion of local confluence: it expresses the contraction of a single redex, but offers little control over how distinct reductions may interact when applied in different orders.

Conversely, parallel reduction is the “maximal” single step of computation: each redex in a is reduced simultaneously and independently of the others, so no order of evaluation is privileged. The hope is that this coarser notion of reduction resolves this problem, smoothing over the interference between overlapping redexes.

It turns out that this hope is realised:

Theorem 3.9 (Local Confluence). *If $t \longrightarrow u$ and $t \longrightarrow v$, then there exists a term s such that $u \longrightarrow s$ and $v \longrightarrow s$:*

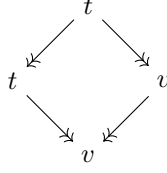


Proof. By induction on the derivation of $t \longrightarrow u$. We consider the last rule in the derivation:

- (var^{\parallel}) :

$$\frac{}{x \longrightarrow x} (\text{var}^{\parallel})$$

Then, $t = x = u$, so by the reflexivity of parallel reduction on v , $s = v$ suffices:



The same argument applies to all other situations where one of the two reductions is reflexive, so ignore such trivial reductions in the remaining cases.

- $(\rightarrow_I^{\parallel})$:

$$\frac{r \longrightarrow u'}{t = \lambda x.r \longrightarrow \lambda x.u' = u}$$

For the other reduction $t \longrightarrow v$, the only rule which non-trivially reduces an abstraction is $(\rightarrow_I^{\parallel})$, so inversion gives $v = \lambda x.v'$ with $r \longrightarrow v'$.

By the inductive hypothesis applied to $r \longrightarrow u'$ and $r \longrightarrow v'$, there exists s' with $u' \longrightarrow s'$ and $v' \longrightarrow s'$. Then, take $s = \lambda x.s'$. By $(\rightarrow_I^{\parallel})$, we obtain $u = \lambda x.u' \longrightarrow \lambda x.s' = s$ and $v = \lambda x.v' \longrightarrow \lambda x.s' = s$.

- $(\rightarrow_E^{\parallel})$:

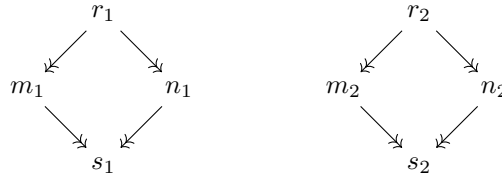
$$\frac{r_1 \longrightarrow m_1 \quad r_2 \longrightarrow m_2}{t = r_1 r_2 \longrightarrow m_1 m_2 = u}$$

The second derivation $t \longrightarrow v$ can then either be:

- $(\rightarrow_E^{\parallel})$, with $r_1 \longrightarrow n_1$ and $r_2 \longrightarrow n_2$, giving $v = n_1 n_2$:

$$\frac{r_1 \longrightarrow n_1 \quad r_2 \longrightarrow n_2}{t = r_1 r_2 \longrightarrow n_1 n_2 = v}$$

By the inductive hypothesis on $r_i \longrightarrow m_i$ and $r_i \longrightarrow n_i$, there exist s_i with:



Let $s = s_1 s_2$. By $(\rightarrow_E^{\parallel})$, we have $u = m_1 m_2 \longrightarrow s_1 s_2 = s$ and $v = n_1 n_2 \longrightarrow s_1 s_2 = s$.

- $(\rightarrow_\beta^{\parallel})$: with $r \longrightarrow n_1$ and $r_2 \longrightarrow n_2$, giving $v = n_1[n_2/x]$:

$$\frac{r \longrightarrow n_1 \quad r_2 \longrightarrow n_2}{t = (\lambda x.r)r_2 \longrightarrow n_1[n_2/x] = v}$$

where $r_1 = \lambda x.r$. There are two cases, depending on the form of m_1 (from the original derivation of $t \longrightarrow u$):

- * If m_1 is an abstraction $\lambda x.m$, then inversion on $r_1 = \lambda x.r \longrightarrow \lambda x.m = m_1$ yields $r \longrightarrow m$. The inductive hypothesis on $r \longrightarrow n_1$ and $r \longrightarrow m$ then obtains a common reduct s_1 of n_1 and m ; and on $r_2 \longrightarrow n_2$ and $r_2 \longrightarrow m_2$ obtains a common reduct s_2 of n_2 and m_2 .

Let $s = s_1[s_2/x]$. Then, $u = m_1 m_2 = (\lambda x.m)m_2 \longrightarrow s_1[s_2/x] = s$ by $(\rightarrow_\beta^{\parallel})$, and $v = n_1[n_2/x] \longrightarrow s_1[s_2/x] = s$ by the substitution lemma.

- * If m_1 is not an abstraction, then $u = m_1 m_2$ is not a β -reduct, so we can apply the inductive hypothesis to the components as in the first subcase to obtain a common reduct $s = s_1 s_2$.

The strategy in both branches is that the inductive hypothesis, along with compatibility of parallel reduction with substitution, allows us to realign any β -reductions performed on one side with the parallel reductions on the other side.

- $(\rightarrow_\beta^\parallel)$:

$$\frac{r \twoheadrightarrow m_1 \quad r_2 \twoheadrightarrow m_2}{t = (\lambda x.r)r_2 \twoheadrightarrow m_1[m_2/x] = u}$$

The second derivation $t \twoheadrightarrow v$ can then either be:

- $(\rightarrow_E^\parallel)$:

$$\frac{r_1 \twoheadrightarrow n_1 \quad r_2 \twoheadrightarrow n_2}{t = r_1 r_2 \twoheadrightarrow n_1 n_2 = v}$$

where $r_1 = \lambda x.r$. By inversion on $r_1 = \lambda x.r \twoheadrightarrow n_1$, we have $n_1 = \lambda x.n$ with $r \twoheadrightarrow n$. The induction hypothesis on $r \twoheadrightarrow m_1$ and $r \twoheadrightarrow n$ obtains a common reduct s_1 of m_1 and n ; and on $r_2 \twoheadrightarrow m_2$ and $r_2 \twoheadrightarrow n_2$ yields a common reduct s_2 of m_2 and n_2 .

Let $s = s_1[s_2/x]$. Then, $u = m_1[m_2/x] \twoheadrightarrow s_1[s_2/x] = s$ by the substitution lemma, and $v = n_1 n_2 = (\lambda x.n)n_2 \twoheadrightarrow s_1[s_2/x]$ by $(\rightarrow_\beta^\parallel)$.

- $(\rightarrow_\beta^\parallel)$:

$$\frac{r \twoheadrightarrow n_1 \quad r_2 \twoheadrightarrow n_2}{t = (\lambda x.r)r_2 \twoheadrightarrow n_1[n_2/x] = v}$$

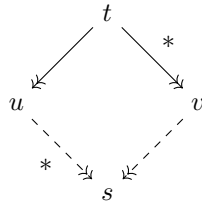
By the inductive hypothesis on $r \twoheadrightarrow m_1$ and $r \twoheadrightarrow n_1$, we obtain a common reduct s_1 ; and on $r_2 \twoheadrightarrow m_2$ and $r_2 \twoheadrightarrow n_2$ a common reduct s_2 . Then $s = s_1[s_2/x]$ suffices, as above.

- Products and sums. The arguments for the remaining product and sum introduction/elimination/ β -reduction rules are entirely analogous to the cases for functions above: apply inversion to the second derivation, apply the inductive hypothesis to resulting subderivations to obtain common reducts of the subterms, and then reassemble with the appropriate parallel-constructors.

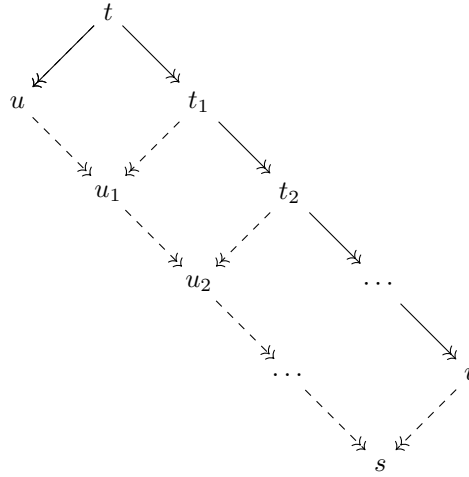
■

From this result, we can easily deduce parallel confluence in two steps. First, as with \rightarrow , we write \twoheadrightarrow^* for the reflexive transitive closure of \twoheadrightarrow .

Lemma 3.10 (Parallel Semi-Confluence). *If $t \twoheadrightarrow u$ and $t \twoheadrightarrow^* v$, then there exists a term s such that $u \twoheadrightarrow^* s$ and $v \twoheadrightarrow^* s$:*

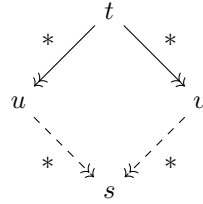


Proof. By induction on the length of the upper-right reduction $t \twoheadrightarrow^* v$, using local confluence of parallel reduction:



■

Theorem 3.11 (Parallel Confluence). *If $t \xrightarrow{*} u$ and $t \xrightarrow{*} v$, then there exists a term s such that $u \xrightarrow{*} s$ and $v \xrightarrow{*} s$:*



Proof. By induction on the length of the upper-left reduction $t \longrightarrow u$, using the previous result. ■

This proves that any two sequences of *parallel* reductions eventually reconverge, but our original goal was to prove this property for sequences of ordinary reductions. Fortunately, every sequence of parallel reductions can be simulated by a sequence of single-step reductions; that is, \longrightarrow^* and \longrightarrow describe precisely the same relation.

Firstly, one-step reductions are special cases of parallel reductions:

Lemma 3.12. *If $t \longrightarrow t'$, then $t \longrightarrow^* t'$.*

Proof. Parallel reduction can reduce any subset of redexes in t ; in particular, it can reduce the single redex used to reduce t to u in a single step. ■

Conversely, parallel reductions are special cases of multi-step reductions:

Lemma 3.13. *If $t \longrightarrow^* t'$, then $t \longrightarrow t'$.*

Proof. By induction on the derivation of $t \longrightarrow^* t'$; the set of redexes reduced by parallel reduction can be reduced in serial by a sequence of one-step reductions. ■

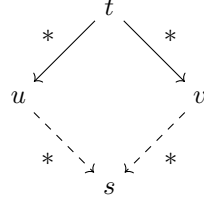
Together, these yield the required equivalence:

Theorem 3.14. *$t \longrightarrow^* t'$ if and only if $t \longrightarrow t'$.*

Proof. If $t \xrightarrow{*} t'$, then t reduces to t' via a sequence of single-step reductions. By the previous lemma, each of these single-step reductions can be replaced by a parallel reduction, so $t \xrightarrow{*} t'$. Conversely, if $t \xrightarrow{*} t'$, then t reduces to t' via a sequence of parallel reductions. By the previous results, each of these parallel reductions can be simulated by a multi-step reduction, so $t \xrightarrow{*} t'$. ■

From this, we can finally deduce the confluence of our simple type theory:

Theorem 3.15 (Confluence). *If $t \xrightarrow{*} u$ and $t \xrightarrow{*} v$, then there exists a term s such that $u \xrightarrow{*} s$ and $v \xrightarrow{*} s$:*



Proof. By transport along the equivalence of $\xrightarrow{*}$ and $\xrightarrow{*}$. ■

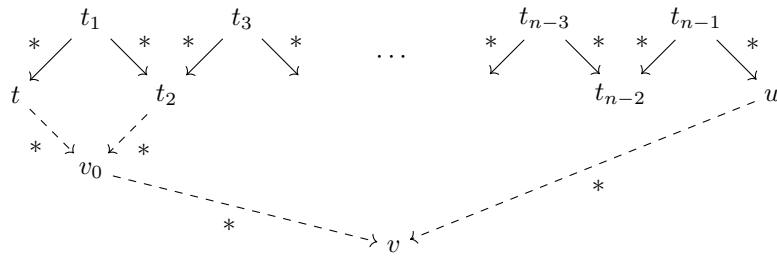
Corollary 3.15.1 (Church-Rosser). *If $t \equiv_{\beta} u$, then there exists a term v such that $t \xrightarrow{*} v$ and $u \xrightarrow{*} v$.*

At first glance, this may appear tautological; by definition, t and u are β -convertible if there is a chain of multi-step reductions connecting them, so perhaps we could take any term in that chain to be v . The problem is that convertibility is the *symmetric* closure of $\xrightarrow{*}$, so $t \equiv_{\beta} u$ is given by a *zig-zag* of forward and backward reduction steps; *a priori*, the witnessing chain may contain arrows in both directions:

$$t \xrightarrow{*} t_1 \xleftarrow{*} t_2 \xrightarrow{*} t_3 \xleftarrow{*} \cdots \xrightarrow{*} t_{n-1} \xleftarrow{*} u$$

The content of Church-Rosser is that one can replace such a zig-zag by a unidirectional reduction: both endpoints reduce (possibly by several steps) to a common reduct v .

Proof. By induction on the number of direction reversals in the chain of reductions connecting t to u . If $u \xrightarrow{*} t$ and $u \xrightarrow{*} t$ directly, then this is just confluence. Otherwise:



where the square on the left is by confluence, and the remainder of the reduction by the inductive hypothesis.

If the sequence starts with an outbound reduction instead, the argument is identical, just with confluence applying to the next shifted square, $t_2 \xrightarrow{*} t_1$ and $t_2 \xrightarrow{*} t_3$ instead, and the reduction is given by $t \xrightarrow{*} t_1 \xrightarrow{*} v_0 \xrightarrow{*} v$. ■

Corollary 3.15.2. *If t and u are β -convertible normal forms, then $t = u$.*

Proof. By Church-Rosser, there exists v such that $t \xrightarrow{*} v$ and $u \xrightarrow{*} v$. But t and u are normal and cannot reduce, so $t = v = u$. ■

Confluence implies the syntactic uniqueness of normal forms: normal reduction constitutes a function from well-typed terms to their normal forms, rather than a mere relation. Moreover, strong normalisation implies that this function is *total* for well-typed terms. We may therefore regard β -convertibility as the canonical and unambiguous notion of equality freely generated by these operational semantics.

In particular, the equivalence of judgemental equality and reductive equality also follows immediately from confluence and strong normalisation:

Theorem 3.16 (Equivalence of Judgemental Equality and Convertibility). *For any two well-typed terms $\Gamma \vdash t : A, u : A$,*

$$\Gamma \vdash t \equiv u : A \iff t \equiv_{\beta} u$$

The forward direction is also called the *soundness* of judgemental equality, and the reverse direction the *completeness* of judgemental equality.

Proof. For soundness, suppose $\Gamma \vdash t \equiv u : A$. We induct on the derivation of $\Gamma \vdash t \equiv u : A$.

The rules for judgemental equality include reflexivity, symmetry, transitivity, and congruence. The first three preserve convertibility since \equiv_{β} is also an equivalence relation. For congruence, if subterms are convertible then the containing terms are convertible, since primitive reduction steps $t \longrightarrow t'$ are also congruent and closed under the β rules by definition. Thus, $t \equiv_{\beta} u$.

For completeness, suppose $t \equiv_{\beta} u$. As t and u are well-typed, they strongly normalise to normal forms \hat{t} and \hat{u} . By confluence, these normal forms are unique, so in fact $\hat{t} = \hat{u}$.

Since judgemental equality is closed under reduction, subject reduction on $t \xrightarrow{*} \hat{t}$ gives a chain of well-typed equalities ending in $\Gamma \vdash t \equiv \hat{t} : A$, and similarly, $u \xrightarrow{*} \hat{u}$ yields $\Gamma \vdash u \equiv \hat{u} : A$. Since $\hat{t} = \hat{u}$, we have by transitivity $\Gamma \vdash t \equiv u : A$.

Hence judgemental equality coincides with reductive equality on well-typed terms. ■

From a proof-theoretic perspective, this equivalence bridges the gap between the syntactic rules governing judgemental equality and the more extensional view offered by reduction. Rather than reasoning about equality through potentially complex derivations, we may instead use reduction to reason about equality, confident that reduction does not identify any more or fewer terms than those identified by the equational theory introduced by the typing rules.

This reduction to normal forms thus acts as a canonicalisation procedure, providing a uniform representation for each equivalence class of terms under β -conversion. In particular, normalisation is *computable* since the reduction relation is both finitely branching and terminating, so we can effectively compute the unique normal form of any well-typed term by systematically applying the reduction rules until no further reductions are possible. In other words, the normalisation function is recursive (in fact, primitive recursive in our formulation).

Since normalisation is computable, we can easily determine when two terms are β -convertible simply by computing their normal forms and comparing them for syntactic equality. Thus:

Theorem 3.17. *Reductive equality is decidable.*

3.3 Unification

Apart from equality, there are a few more algorithmic problems that we consider in type theory:

- *Type checking* – given a context Γ , a term t , and a type A , decide whether $\Gamma \vdash t : A$ is derivable;
- *Type inference* – given a context Γ and a well-typed term t , reconstruct the type A such that $\Gamma \vdash t : A$ is derivable;
- *Typability* – given a term t , decide whether there exists a context Γ and type A such that $\Gamma \vdash t : A$ is derivable;

Confluence and strong normalisation also provide a basis for the decidability of type checking and inference.

Given a context, term, and type, we can mechanically reduce the term to normal form 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 given type. Since strong normalisation and confluence guarantees termination and determinacy, this yields a finite algorithm to determine whether a term inhabits a given type, so type checking is decidable.

Similarly, type inference is also computable, since we can compute a given term's normal form and identify its type by inspecting the syntactic structure of this canonical representative.

While the decidability of judgemental equality, type checking, and type inference all rely on the computational properties of reduction, the situation is somewhat different for typability. These previous results apply only to already well-typed terms, so confluence and strong normalisation guarantee that equality and typing can be verified by computation *within* the calculus. However, for typability, we are not given the typing context or the result type in advance, so the question is no longer reducible to normalisation alone, and it may be the case that the term is in fact not typable in the first place. Instead, this problem requires a syntactic reconstruction of typing information directly from the structure of the term itself.

This problem is much more complex than plain type inference because we also need to determine types for all free variables in t to generate the context Γ before we can attempt to construct the result type; and moreover, these free variables may interact with each other in non-trivial ways.

This can be done by traversing the syntax of the term, and assigning provisional *type variables* to all free variables encountered; each constructor then enforces a finite collection of *type constraints* on these type variables.

For instance, in an application tu , we introduce fresh type variables

$$t : \alpha, \quad u : \beta, \quad tu : \gamma$$

and the typing rule for application requires types of the shape $t : A \rightarrow B$ and $u : A$ and yields $tu : B$, so this gives a constraint

$$\alpha = \beta \rightarrow \gamma$$

We can then recursively traverse and assign types to subterms of t and u .

Collecting these equations over all subterms yields a finite system of type constraints between these type variables, called a *unification problem*. The goal of unification is to determine whether there is a consistent type substitution, or *unifier*, σ mapping type variables α_i to types A_i . One standard algorithm for unification is as follows.

Initialise σ to be the empty substitution, and let E be the system of type constraints. At each step, we will remove one equation from E and simplify it according to the following rules, whilst updating σ . We write $E \Rightarrow \sigma$ to denote that system E is unified by substitution σ .

- Deletion – trivial constraints can be removed:

$$E \cup \{\alpha = \alpha\} \rightsquigarrow E$$

- Decomposition:

$$E \cup \{A_1 \rightarrow A_2 = B_1 \rightarrow B_2\} \rightsquigarrow E \cup \{A_1 = B_1\} \cup \{A_2 = B_2\}$$

and similar for the other type formers \times and $+$. If the top-level constructors conflict, then no unifier exists.

- Swap:

$$E \cup \{A = \alpha\} \rightsquigarrow E \cup \{\alpha = A\}$$

- Elimination: if α is not free in A (the *occurs check*), then

$$E \cup \{\alpha = A\} \rightsquigarrow E[A/\alpha]$$

and extend σ by $\alpha \mapsto A$.

If α is free in β , then no unifier exists, since α would appear as a subterm of itself, requiring an infinite type expression to unify.

This algorithm always terminates since the initial system E is finite and each decomposition and elimination step strictly decreases a well-founded measure (the structural depths of the type expressions and the number of unsolved variables, respectively), and infinite expansions are prevented by the occurs check.

Example. Consider the term $\langle \lambda x.yx.y \text{ inl}(z) \rangle$. We recursively assign type variables

$$\begin{aligned} \langle \lambda x.yx.y \text{ inl}(z) \rangle &: \tau \\ \lambda x.yx &: \alpha \\ yx &: \beta \\ x &: \gamma \\ y \text{ inl}(z) &: \delta \\ y &: \varepsilon \\ \text{inl}(z) &: \zeta \\ z &: \eta \\ \bullet &: \vartheta \end{aligned}$$

where ϑ is the right summand type implied by inl . The type constraints are then:

$$\begin{aligned} \tau &= \alpha \times \delta & (\times_I) \\ \alpha &= \gamma \rightarrow \beta & (\rightarrow_I) \\ \varepsilon &= \gamma \rightarrow \beta & (\rightarrow_E) \\ \zeta &= \eta + \vartheta & (+_I) \\ \varepsilon &= \zeta \rightarrow \delta & (\rightarrow_E) \end{aligned}$$

We can eliminate the last constraint to obtain

$$\begin{aligned} \tau &= \alpha \times \delta \\ \alpha &= \gamma \rightarrow \beta \\ \zeta \rightarrow \delta &= \gamma \rightarrow \beta \\ \zeta &= \eta + \vartheta \end{aligned}$$

and adding to the substitution $\varepsilon \mapsto \gamma \rightarrow \beta$. We can then decompose this new function constructor to obtain

$$\begin{aligned} \tau &= \alpha \times \delta \\ \alpha &= \gamma \rightarrow \beta \\ \gamma &= \zeta \\ \beta &= \delta \\ \zeta &= \eta + \vartheta \end{aligned}$$

Eliminating these two new constraints, we have

$$\tau = \alpha \times \delta$$

$$\begin{aligned}\alpha &= \zeta \rightarrow \delta \\ \zeta &= \eta + \vartheta\end{aligned}$$

with $\gamma \mapsto \zeta$ and $\beta \mapsto \delta$. Eliminating the last two equations then obtains

$$\tau = ((\eta + \vartheta) \rightarrow \delta) \times \delta$$

with $\zeta \mapsto \eta + \vartheta$ and $\alpha \mapsto (\eta + \vartheta) \rightarrow \delta$.

So, this term is typable with:

$$y : (\eta + \vartheta) \rightarrow \beta, z : \eta \vdash \langle \lambda x : \eta + \vartheta. yx, y \text{ inl}(z) \rangle : ((\eta + \vartheta) \rightarrow \delta) \times \delta$$

△

Example. Consider the term $\lambda x.xx$. Unlike the Ω combinator, this term is a normal form, so its typability cannot be ruled out by the strong normalisation theorem. However, we will see that the typing constraints force an occurs-check cycle, so the term does not admit a finite simple type. We assign type variables

$$\begin{aligned}\lambda x.xx &: \tau \\ x &: \alpha \\ xx &: \beta\end{aligned}$$

with type constraints

$$\begin{aligned}\tau &= \alpha \rightarrow \beta \\ \alpha &= \gamma \rightarrow \beta \\ \alpha &= \gamma\end{aligned}$$

From $\alpha = \gamma$, we can eliminate γ to obtain

$$\begin{aligned}\tau &= \alpha \rightarrow \beta \\ \alpha &= \alpha \rightarrow \beta\end{aligned}$$

Attempting to eliminate the second equation shows that this term is not typable. △

Together, these results show that our simple type theory is not only syntactically well-defined, but also *algorithmically tractable*: reduction is confluent and strongly normalising, type checking is decidable by normal-form computation, and judgemental equality coincides with a decidable notion of reductive equality.

The calculus is thus both *consistent* — it admits no ill-typed or non-terminating derivations — and *complete* with respect to its own operational semantics, in that every equality and typing judgement can be resolved by a finite syntactic procedure. In this sense, simple type theory constitutes an *effective* formal system — one whose judgements can be mechanically verified or refuted, and whose terms admit a well-behaved notion of computation that are guaranteed to terminate.

4 The Curry-Howard Correspondence

Up to this point, our attention has been directed toward the syntactic and operational properties of our simple type theory; we 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 compositional patterns that are reminiscent of *semantic reasoning*.

4.1 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:

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} (\rightarrow_E)$$

we only require a value typed by the function type $A \rightarrow B$, 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:

$$\frac{\Gamma \vdash A \rightarrow B, \quad \Gamma \vdash A}{\Gamma \vdash B}$$

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:

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

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 :

$$\begin{array}{ccc} \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B} (\times_I) & & \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} (\wedge_I) \\ \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \text{fst}(p) : A} (\times_{E_1}) \quad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \text{snd}(p) : B} (\times_{E_2}) & \rightsquigarrow & \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} (\wedge_{E_1}) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} (\wedge_{E_2}) \end{array}$$

- Sums $+$ correspond to disjunction \vee :

$$\begin{array}{ccc} \frac{\Gamma \vdash a : A}{\Gamma \vdash \text{inl}(a) : A + B} (+_{I_1}) \quad \frac{\Gamma \vdash b : B}{\Gamma \vdash \text{inr}(a) : A + B} (+_{I_2}) & & \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} (\vee_{E_1}) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} (\vee_{I_2}) \\ \frac{\Gamma \vdash t : A + B \quad \Gamma \vdash f : A \rightarrow C \quad \Gamma \vdash g : B \rightarrow C}{\Gamma \vdash \text{case}(t; f, g) : C} (+_E) & \rightsquigarrow & \frac{\Gamma \vdash A \vee B \quad \Gamma \vdash A \Rightarrow C \quad \Gamma \vdash B \Rightarrow C}{\Gamma \vdash C} (\vee_I) \end{array}$$

- The unit type corresponds to tautology \top :

$$\begin{array}{ccc} \frac{}{\Gamma \vdash t : \text{unit}} (\text{unit}_I) & & \frac{}{\Gamma \vdash \top} (\top_I) \\ \frac{\Gamma \vdash t : \text{unit} \quad \Gamma \vdash b : B}{\Gamma \vdash B} (\text{unit}_E) & \rightsquigarrow & \frac{\Gamma \vdash \top \quad \Gamma \vdash B}{\Gamma \vdash B} (\top_E) \end{array}$$

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 – that the assumption \top carries no data and does not contribute anything in the proof of B .

- The `void` type corresponds to contradiction \perp :

$$\frac{\Gamma \vdash t : \text{void}}{\Gamma \vdash B}(\text{void}_E) \quad \rightsquigarrow \quad \frac{\Gamma \vdash \perp}{\Gamma \vdash B}(\perp_E)$$

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*.

Theorem 4.1 (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 the typing sequent $\Gamma \vdash t : A$ is a valid intuitionistic proof of the logical sequent $\Gamma \vdash A$.

Conversely, given a proof of a logical sequent, we can construct a term with this proof as a typing derivation by induction on the proof derivation:

- If the last rule is axiom:

$$\overline{\Gamma, A, \Delta \vdash A}$$

then we may use the axiom rule for typing:

$$\overline{\Gamma, t : A, \Delta \vdash t : A}$$

- If the last rule is (\Rightarrow_E) :

$$\frac{\frac{\pi_1}{\Gamma \vdash A \Rightarrow B} \quad \frac{\pi_2}{\Gamma \vdash A}}{\Gamma \vdash B}$$

then by the inductive hypothesis we have terms t and u with typing derivations

$$\frac{\pi'_1}{\Gamma \vdash t : A \rightarrow B} \quad \frac{\pi'_2}{\Gamma \vdash u : A}$$

so by (\rightarrow_I) ,

$$\frac{\frac{\pi'_1}{\Gamma \vdash t : A \rightarrow B} \quad \frac{\pi'_2}{\Gamma \vdash u : A}}{\Gamma \vdash t u : B}$$

- If the last rule is (\Rightarrow_I) :

$$\frac{\frac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}$$

then by the inductive hypothesis we have a typing derivation

$$\frac{\pi'}{\Gamma, x : A \vdash t : B}$$

so by (\rightarrow_I) :

$$\frac{\frac{\pi'}{\Gamma, x : A \vdash t : B}}{\Gamma \vdash \lambda x. t : A \Rightarrow B}$$

- The other rules for products and sums are similar.

■

Under this correspondence, we can interpret well-typed terms as *proofs* of the propositions their types represent. This interpretation naturally extends to all of the compound type formers we have seen so far:

- A proof of $A \Rightarrow B$ is a function that transforms a proof of A into a proof of B ;
- A proof of a product $A \wedge B$ is a proof of A , paired with a proof of B ;
- A proof of a sum $A \vee B$ is either a proof of A , or a proof of B , *along with a tag indicating which is the case*;
- A proof of $\neg A \equiv A \rightarrow \perp$ is a function that derives a proof of \perp from a purported proof of A .

We can then prove propositions by constructing terms that inhabit their corresponding types.

For example, the proposition $A \Rightarrow (B \Rightarrow A)$ is provable because $A \rightarrow (B \rightarrow A)$ is inhabited by the term $\lambda a.\lambda b.a$. In this proof interpretation, we read this as the function that takes a proof a of A , and returns the constant function $\lambda b.a$ that takes a proof b of B and returns the original given proof a of A .

Our disjunctions being tagged is one of the reasons our logic is intuitionistic. In classical logic, it may not be possible to extract a proof of A or B from a proof of $A \vee B$. In contrast, this is always possible in our current system, since a proof of $A \vee B$ is given explicitly by a proof of A or a proof of B . In this way, our proofs carry more information than classical ones.

On the other hand, we cannot prove any non-constructive principles. For instance, there is, *in general*, no way of constructing a term of type $P \vee \neg P$ (LEM) or $\neg\neg P \rightarrow P$ (DNE) for an arbitrary proposition P :

- By strong normalisation and canonical forms, a closed term of type $P \vee \neg P \equiv P + (P \rightarrow \text{void})$ reduces to either $\text{inl}(p)$ for some $p : P$ or $\lambda x.t$ for some $t : \text{void}$ in context $x : P$. Unless we have a constructor for P , we don't have a way of constructing such a term for a generic type P , and we don't specifically don't have a constructor for void .
- Similarly, a closed term of type $\neg\neg P \equiv ((P \rightarrow \text{void}) \rightarrow \text{void}) \rightarrow P$ reduces to a λ -abstraction $\lambda x.t$, where $x : (P \rightarrow \text{void}) \rightarrow \text{void}$, and $t : P$ has no free variables, apart possibly from x . For a generic type P , we have no knowledge of any constructors, so t cannot be an introduction form for P , and must therefore be a term whose head is the free variable x . However, any such term has a type obtained by starting from the type of x and eliminating successive argument arrows, so only $(P \rightarrow \text{void}) \rightarrow \text{void}$ and void are possible, which in general are not P .

The problem is that classical logic is *proof-irrelevant*. Classically, the distinction between different proofs of the same proposition is semantically invisible: all proofs of a true proposition are interchangeable, and all refutations of a false one are equivalent to the unique proof of absurdity. The logic thus reduces propositions to their mere truth values, abstracting away any notion of computational or constructive content.

However, a shadow of this proof-irrelevance can be seen in intuitionistic logic too. We interpret the double negation $\neg\neg A \equiv (A \rightarrow \perp) \rightarrow \perp$ as a function that takes any purported function that transforms a proof of A into a proof of \perp , and returns a proof of \perp . In other words, $\neg\neg A$ is a proof that A cannot be disproven. However, this does not, in general, constitute a proof of A .

In particular, under the proof interpretation:

- if A is inhabited (i.e. A is provable), then $\neg A = \{A \rightarrow \emptyset\}$ is empty;
- while if A is empty, then $\neg A = \{\emptyset \rightarrow \emptyset\}$ has exactly one proof – the empty function.

Now, applying this reasoning to $\neg A$:

- if A is inhabited, then $\neg\neg A = \{\neg A \rightarrow \emptyset\} = \{\emptyset \rightarrow \emptyset\}$ has precisely one term;
- if A is empty, then $\neg\neg A = \{\neg A \rightarrow \emptyset\}$ is empty.

Regardless of how many proofs of A we have, as long as we have *at least* one, there is only one proof of $\neg\neg A$! We can interpret the double negation of A as a kind of *existential* claim about A : $\neg\neg A$ is provable if there *exists* a proof of A .

In light of this, the classical axiom *Double Negation Elimination* DNE

$$\vdash \neg\neg A \Rightarrow A$$

is then some dark magic that extracts a proof of A from this information-free existential claim. The only way this can be tenable is if all of these types have at most one term; that is, if all proofs are definitionally equal and carry no constructive content: classical logic is proof-irrelevant.

This is precisely the collapse to a *Boolean* proof theory that is characteristic of classical logic when interpreted naively in this way: propositions become plain *truth values*, and proofs carry no structure beyond mere existence. Consequently, the proof theory of classical logic is degenerate in the sense that there is no room for a finer analysis of proofs or of their computational content.

In contrast, the intuitionistic logic our type theory corresponds to directly rejects this collapse: proofs – that is, well-typed terms – in this system can carry rich structure and two proofs of the same proposition

need not be identified. The result is a constructive proof theory that does not collapse to Boolean truth values; a constructive proof does not merely assert that something is the case, but provides an explicit witness demonstrating it, and intuitionistic reasoning preserves information about *how* a result is established, rather than merely whether it holds or not.

We can nonetheless recover a connection between classical and intuitionistic reasoning through a systematic translation that embeds classical logic into our intuitionistic setting. We can simulate the proof-irrelevant structure of classical logic by systematically shielding our propositions under double negation, which captures classical proof-irrelevance intuitionistically.

We encode each classical formula A by an intuitionistic formula A^* whose provability in intuitionistic logic corresponds to the classical provability of A .

$$\Gamma \vdash_{\text{LK}} A \iff \Gamma \vdash_{\text{LJ}} A^*$$

This translation, known as the *double negation translation* or *Gödel—Gentzen negative translation*, provides a way of embedding classical logic into intuitionistic. One convenient formulation of the translation is given inductively by:

$$\begin{aligned} P^* &:= \neg\neg P && (P \text{ atomic}) \\ (A \wedge B)^* &:= A^* \wedge B^* \\ (A \vee B)^* &:= \neg(\neg A^* \wedge \neg B^*) \\ (A \rightarrow B)^* &:= A^* \rightarrow B^* \\ (\neg A)^* &:= \neg A^* \end{aligned}$$

Under this translation, a sequent $\Gamma \vdash A$ is provable in classical sequent calculus (LK) if and only if the translated sequent $\Gamma \vdash A^*$ is provable in intuitionistic sequent calculus (LJ). In this way, intuitionistic logic is not weaker, but more *discerning*: it distinguishes between the raw existence of a proof (classical provability) and the constructive realisation of one (intuitionistic provability). From this perspective, classical logic is not a system incompatible with intuitionistic reasoning, but is rather its degenerate case, obtained by universally erasing the data carried by proofs.

4.2 Cut and β -Reduction

So far, we have seen that introduction and elimination rules correspond to the rules of intuitionistic sequent calculus, but what about β -reduction?

Consider the (\times_{β_1}) reduction rule:

$$\text{fst}\langle a, b \rangle \longrightarrow a$$

If we write out the derivation of the left term in full, we have:

$$\frac{\frac{\frac{\pi}{\Gamma \vdash a : A} \quad \frac{\pi'}{\Gamma \vdash b : B}}{\Gamma \vdash \langle a, b \rangle : A \times B} (\times_I)}{\Gamma \vdash \text{fst}\langle a, b \rangle : A} (\times_E)$$

Erasing terms, we have the derivation:

$$\frac{\frac{\frac{\pi}{\Gamma \vdash A} \quad \frac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B} (\wedge_I)}{\Gamma \vdash A} (\wedge_E)$$

Viewed as a logical derivation, this proof seems to have some unnecessary steps: this is a proof of A , but the way we get there is to first start with a proof a of A in the first place, pair it with a proof b of B , then project the proof of A back out! To prove A , the proof a already suffices, and the proof b of B is completely superfluous.

In fact, this is precisely what the reduction rule says – it eliminates the detour through the product and just returns the proof a of A :

$$\frac{\pi}{\Gamma \vdash a : A}$$

Similarly, for the (\rightarrow_β) rule,

$$(\lambda x : A. t)u \longrightarrow t[u/x]$$

we have the corresponding derivation:

$$\frac{\frac{\frac{\pi}{\Gamma, x : A \vdash t : B}}{\Gamma \vdash \lambda x. t : A \rightarrow B} (\Rightarrow_I) \quad \frac{\pi'}{\Gamma \vdash u : A} (\Rightarrow_E)}{\Gamma \vdash (\lambda x. t) u : B} (\Rightarrow_E)$$

In this derivation, we have shown in π that we can prove B under the assumption of a proof of A . Then, we discharge this assumption to obtain the implication $A \rightarrow B$, from which we deduce B by supplying a proof of A . As before, we could have just directly proved B by replacing in π all the instances of the hypothesis A by π' .

This is again exactly what the reduction rule says:

$$\frac{\pi[\pi'/x]}{\Gamma \vdash t[u/x]}$$

This redundant logical structure given by a redex is called a *cut*. Such inefficient structures may seem rather pointless at first, but this line of deduction is actually very common in natural reasoning; we can prove a result by first proving a more general lemma, then deducing the target result from there. While this may involve extra work in some cases, this modularity allows us to reuse the lemma in multiple places, possibly resulting in a shorter proof overall.

For instance, if the hypothesis $x : A$ is used extensively in π , and/or π' is very long, then replacing every instance of x with π' could be a very expensive operation resulting in a much longer proof overall, even though we ostensibly removed a “detour”. Viewed another way, the implication cuts correspond to *proof composition*, where we insert the proof of a lemma A into a proof of B .

A logical calculus has the *cut elimination property* if, whenever a formula is provable, it also admits a proof that does not contain any cuts.

As noted above, when eliminating a cut, this replacement procedure could duplicate the second proof π' – which itself may contain cuts – many times, so it is not at all obvious that naïvely eliminating cuts will ever terminate.

However, via the Curry-Howard correspondence, β reduction corresponds precisely to *cut elimination*, and this result immediately follows:

Theorem 4.2. *Intuitionistic propositional logic has the cut elimination property.*

Proof. By the Curry-Howard correspondence, a cut-free proof is precisely a closed term in normal form. As our type theory is strongly normalising, every closed term reduces to a normal form, so every proof admits a cut-free proof. ■

A system that admits the cut elimination property can also easily be shown to be consistent, since any proof of \perp can be transformed into a cut-free proof, and it is usually easy to verify that there are no such proofs by checking through all of the introduction rules.

Under the Curry-Howard correspondence, this proof manifests as strong normalisation and canonical forms:

Theorem 4.3. *If a type system satisfies strong normalisation, then it is consistent. That is, there is no closed term of type `void`.*

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

4.3 Program = Proof

The correspondence between cut elimination and β -reduction also connects proofs with our earlier operational interpretation of terms: proofs are *programs* that can be run. This interpretation is perhaps hard to see with only the opaque or trivial base types we have so far, so we introduce a type that is more amenable to a computational interpretation.

The `nat` type represents the natural numbers, and is defined from a base constant 0, and a successor function `succ` : `nat` \rightarrow `nat`:

- Introduction:

$$\frac{}{\Gamma \vdash 0 : \text{nat}} \quad \frac{\Gamma \vdash n : \text{nat}}{\Gamma \vdash \text{succ}(n) : \text{nat}}$$

that is, 0 is always a term of type `nat`, and given a term n of type `nat`, its *successor* `succ`(n) is also a term of type `nat`.

- Elimination: to eliminate a term of type `nat`, we provide an initial value $b : B$ and a recursion function $r : B \rightarrow B$ for successors:

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

- Computation:

$$\begin{aligned} \text{natrec}(r, b, 0) &\equiv b \\ \text{natrec}(r, b, \text{succ}(n)) &\equiv r(\text{natrec}(r, b, n)) \end{aligned}$$

Intuitively, the input $b : B$ corresponds to the result of eliminating the base case 0, while r tells us how to eliminate `succ`(n) in terms of n . 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 `succ`(n) for some $n : \text{nat}$. In particular, $0 \not\equiv \text{succ}(m)$ for all $m : \text{nat}$, and `succ` is injective.

Let us implement addition

$$\text{add} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$$

using the elimination rule.

Here, the output type is $B \equiv \text{nat} \rightarrow \text{nat}$. We give $b : \text{nat} \rightarrow \text{nat}$ as

$$b \equiv \lambda n. n$$

since adding zero should be the identity map, and $r : (\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat})$ by

$$r \equiv \lambda f. \lambda m. \text{succ}(f m)$$

where f will be instantiated by the partially-applied addition function $n + (-)$. The final expression is then given by:

$$\text{add} \equiv \lambda m. \lambda n. \text{natrec}(b, r, m) n$$

The computation rules are then:

$$\begin{aligned}\text{add}(0,n) &\equiv \text{natrec}(b,r,0) \, n \\ &\equiv b \, n \\ &\equiv n \\ \text{add}(\text{succ}(m),n) &\equiv \text{natrec}(b,r,\text{succ}(m)) \, n \\ &\equiv r(\text{natrec}(b,r,m)) \, n \\ &\equiv \text{succ}(\text{add}(m,n))\end{aligned}$$

which are precisely the Peano axioms for addition.

5 Inductive Types

5.1 nat Revisited

5.2 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 :

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

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 $T \rightarrow B$, 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 without any assumptions:

$$\overline{T \text{ type}}$$

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

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

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 $f : T \rightarrow B$ by specifying its behaviour on each constructor, including recursively built terms.

Formally, for each constructor

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

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

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

where

$$(A_{i,j})' := A_{i,j}[B/T]$$

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 $f(c_i(\dots))$.

The *recursor*

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

is then introduced by:

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

(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:

$$\text{rec}_T^B(f_1, \dots, f_k)(c_i(a_1, \dots, a_{n_i})) \equiv f_i(a'_1, \dots, a'_{n_i})$$

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 $\text{lift}_X : (T \rightarrow B) \rightarrow X \rightarrow X[B/T]$ by:

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

where lift_X recurses over all of the constructors of X .

Then, we transform inputs to the step functions as:

$$a'_j \equiv \begin{cases} a_j & A_{i,j} \text{ does not involve } T \\ \text{lift}_{A_{i,j}}(\text{rec}_T^B(f_1, \dots, f_k))(a_j) & \text{otherwise} \end{cases}$$

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

The reason we cannot immediately replace a_j by $\text{rec}(a_j)$ 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:

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

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

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

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 5.1 (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 $\text{Bool} \equiv \text{unit} + \text{unit}$ be the type with two inhabitants, labelled \top and \perp . We define a function $f : T \rightarrow \text{Bool}$ with the constant step functions

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

and

$$f_j \equiv \lambda b_1 \dots b_{n_j}. \perp$$

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

$$f(c_i(a_1, \dots, a_{n_i})) \equiv \top, \quad f(c_j(b_1, \dots, b_{n_j})) \equiv \perp$$

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 $\top \equiv \perp$, which is a contradiction, as \top and \perp are distinct terms of Bool . \blacksquare

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

$$\begin{aligned} \text{inl}(a_1) \equiv \text{inl}(a_2) &\implies a \equiv x \\ \text{inr}(b_1) \equiv \text{inr}(b_2) &\implies b \equiv y \end{aligned}$$

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

$$\begin{aligned} f_1 &\equiv \lambda x. x \\ f_2 &\equiv \lambda y. a_0 \end{aligned}$$

By the computation rules,

$$\begin{aligned} \text{rec}_{A+B}^A(f_1, f_2)(\text{inl}(a_1)) &\equiv f_1(a_1) \equiv a_1 \\ \text{rec}_{A+B}^A(f_1, f_2)(\text{inl}(a_2)) &\equiv f_1(a_2) \equiv a_2 \end{aligned}$$

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

$$\text{rec}_{A+B}^A(f_1, f_2)(\text{inl}(a_1)) \equiv \text{rec}_{A+B}^A(f_1, f_2)(\text{inl}(a_2))$$

and hence $a_1 \equiv a_2$. ■

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

Theorem 5.3 (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

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

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 i th sum injection:

$$f_i(x'_1, \dots, x'_{n_i}) \equiv \text{in}_i(\langle x'_1, \dots, x'_{n_i} \rangle)$$

Then, by the computation rule, we have:

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

By assumption,

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

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

$$\text{in}_i(\langle a'_1, \dots, a'_{n_i} \rangle) \equiv \text{in}_i(\langle b'_1, \dots, b'_{n_i} \rangle)$$

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 $(-)' \equiv \text{lift}(\text{rec})$ 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, $A_{i,j}[B/T] \equiv B$,

$$\begin{aligned} a'_j &\equiv \text{rec}_T^B(f_1, \dots, f_k)(a_j) \\ b'_j &\equiv \text{rec}_T^B(f_1, \dots, f_k)(b_j) \end{aligned}$$

By the computation rule of rec , and the choice of B and f_i , rec sends each term $c_i(\vec{t})$ to $\text{in}_i(\langle t'_1, \dots, t'_{n_i} \rangle)$ (i.e. the i th 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 ;

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

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

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

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,

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

- 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

$$\begin{aligned} a'_j &\equiv \langle \text{lift}_{X_1}(\text{rec})(a_{j,1}), \text{lift}_{X_2}(\text{rec})(a_{j,2}) \rangle \\ b'_j &\equiv \langle \text{lift}_{X_1}(\text{rec})(b_{j,1}), \text{lift}_{X_2}(\text{rec})(b_{j,2}) \rangle \end{aligned}$$

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

$$\begin{aligned} \text{lift}_{X_1}(h)(a_{j,1}) &\equiv \text{lift}_{X_1}(h)(b_{j,1}) \\ \text{lift}_{X_2}(h)(a_{j,2}) &\equiv \text{lift}_{X_2}(h)(b_{j,2}) \end{aligned}$$

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 $a'_j \equiv \text{inl}(\text{lift}_{X_1}(\text{rec})(x))$ and $b'_j \equiv \text{inl}(\text{lift}_{X_1}(\text{rec})(y))$. By assumption, $a'_j \equiv b'_j$, so

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

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 $\text{lift}_{A_{i,j}}(\text{rec})$ is injective, and hence we obtain $a_j \equiv b_j$ for $j = 1, \dots, n_i$. ■

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:

$$0 : \text{nat} \quad \text{and} \quad \text{succ} : \text{nat} \rightarrow \text{nat}$$

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

$$\frac{\Gamma \vdash n : \text{nat} \quad \Gamma \vdash b : B \quad \Gamma \vdash r : B \rightarrow B}{\Gamma \vdash \text{rec}_{\text{nat}}^B(b, r)(t) : B}$$

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

$$\begin{aligned} \text{natrec}(0) &\equiv b_0 \\ \text{natrec}(\text{succ}(n)) &\equiv r(\text{natrec}(n)) \end{aligned}$$

which is precisely the definition of simple recursion. \triangle

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

- the injectivity of constructors:

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

- and the disjointness of constructors:

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

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,

$$0 \neq \text{succ}(n) \quad \text{and} \quad \text{succ}(n) \equiv \text{succ}(m) \iff n \equiv m$$

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:

$$\star : \text{unit}$$

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

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

The computation rule is then what we had before:

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

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

$$t \equiv \star$$

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

The **void** type is another trivial inductive type, with no constructors. The recursor $\text{abort} = \text{rec}_{\text{void}}^B : \text{void} \rightarrow B$ 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:

$$\frac{\Gamma \vdash t : \text{void}}{\Gamma \vdash \text{rec}_{\text{void}}^B(t) : B}$$

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 $\text{List}(A)$ be the inductive type with constructors:

$$\text{nil}_A : \text{List}(A) \quad \text{and} \quad \text{cons}_A : A \rightarrow \text{List}(A) \rightarrow \text{List}(A)$$

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

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

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

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

$$\begin{aligned} \text{rec}_{\text{List}(A)}^B(b, f)(\text{nil}_A) &\equiv b : B \\ \text{rec}_{\text{List}(A)}^B(b, f)(\text{cons}_A(a, \ell)) &\equiv f(a, \text{rec}_{\text{List}(A)}^B(b, f)(\ell)) : B \end{aligned}$$

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

$$\begin{aligned} f_1 &::= 0 : \text{nat} \\ f_2 &::= \lambda a. \lambda n. \text{succ}(n) : A \rightarrow \text{nat} \rightarrow \text{nat} \end{aligned}$$

Then, $\text{len} ::= \text{rec}_{\text{List}(A)}^{\text{nat}}(f_1, f_2) : \text{List}(A) \rightarrow \text{nat}$ satisfies

$$\begin{aligned} \text{len}(\text{nil}_A) &\equiv 0 : \text{nat} \\ \text{len}(\text{cons}_A(a, \ell)) &\equiv \text{succ}(\text{len}(\ell)) : \text{nat} \end{aligned}$$

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

$$\begin{aligned} f_1 &::= \text{nil}_B : \text{List}(B) \\ f_2 &::= \lambda a. \lambda \ell'. \text{cons}_B(g(a), \ell') : A \rightarrow \text{List}(B) \rightarrow \text{List}(B) \end{aligned}$$

We note that the step function f_2 depends on the function $g : A \rightarrow B$. 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:

$$g : A \rightarrow B \vdash f_2 : A \rightarrow \text{List}(B) \rightarrow \text{List}(B)$$

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

$$\text{map} ::= \lambda g : A \rightarrow B. \text{rec}_{\text{List}(A)}^{\text{List}(B)}(f_1, f_2) : (A \rightarrow B) \rightarrow \text{List}(A) \rightarrow \text{List}(B).$$

This definition satisfies the computation rules:

$$\begin{aligned}\text{map}(g, \text{nil}_A) &\equiv \text{nil}_B : \text{List}(B) \\ \text{map}(g, \text{cons}_A(a, \ell)) &\equiv \text{cons}_B(g(a), \text{map}(g, \ell)).\end{aligned}$$

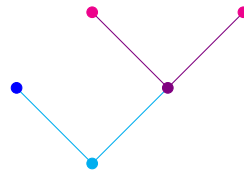
That is, mapping g over the empty A -list returns the empty B -list; and mapping g over the A -list with head a and tail ℓ returns the B -list with head $g(a)$ and tail given by recursively mapping g over ℓ .

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

$$\text{leaf} : T \quad \text{and} \quad \text{node} : T \times T \rightarrow T$$

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

`node(leaf, node(leaf, leaf))`



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:

$$\text{pair} : A \rightarrow B \rightarrow A \times B$$

and recursor satisfying:

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

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

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,

$$\begin{aligned} \text{fst} &\equiv \lambda p. \text{rec}_{A \times B}^A(\lambda a. \lambda b. a)(p) \\ \text{snd} &\equiv \lambda p. \text{rec}_{A \times B}^A(\lambda a. \lambda b. b)(p) \end{aligned}$$

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

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

△

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

$$\text{inl} : A \rightarrow A + B \quad \text{and} \quad \text{inr} : B \rightarrow A + B$$

with recursor satisfying

$$\begin{aligned} \text{rec}_{A+B}^C(f_1, f_2)(\text{inl}(a)) &\equiv f_1(a) \\ \text{rec}_{A+B}^C(f_1, f_2)(\text{inr}(b)) &\equiv f_2(b) \end{aligned}$$

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

△

5.2.1 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 $A \rightarrow B$ 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

$$t_0 : T \quad \text{and} \quad \text{foo} : (A \rightarrow T) \rightarrow T$$

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

$$\text{rec}_T^B : B \rightarrow ((A \rightarrow B) \rightarrow B) \rightarrow T \rightarrow B$$

with computation rules of the form:

$$\begin{aligned} \text{rec}_T^B(b_0, f)(t_0) &\equiv b_0 \\ \text{rec}_T^B(b_0, f)(\text{foo}(g)) &\equiv f(g') \end{aligned}$$

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

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

When constructing an inductive type with a constructor like

$$\text{bar} : A \times T \rightarrow T$$

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

$$\text{foo} : (A \rightarrow T) \rightarrow T$$

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

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* $g : \text{unit} \rightarrow T$ 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:

$$\begin{aligned} \text{lift}_{X_1 \rightarrow X_2}(h) &: (X_1 \rightarrow X_2) \rightarrow (X'_1 \rightarrow X'_2) \\ \text{lift}_{X_1 \rightarrow X_2}(h)(f) &:= \lambda x'. \text{lift} \end{aligned}$$

Even if we tried manually

Since we have no strictly

Suppose we have

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

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

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

If we try to define

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

then

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

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

Now, define

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

Expanding h , we have

$$\Omega \equiv \text{foo}(\lambda x. \text{foo}(\lambda y. x))$$

$$\equiv \text{foo}(\lambda x. \Omega)$$

$$f : T \rightarrow T :\equiv \text{foo}(\lambda x. \Omega)$$

5.3 Derivations

6 Dependent Type Theory

6.1 Judgements

6.2 Universe Types

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

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

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

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

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:

$$\text{Type} : \text{Type}$$

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:

$$\text{Type}_0 : \text{Type}_1 : \text{Type}_2 : \text{Type}_3 : \dots$$

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.

6.3 Dependent Sums

6.4 Dependent Products

6.5 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

$$\frac{}{\Gamma \vdash 0 : \text{nat}} \quad \frac{\Gamma \vdash n : \text{nat}}{\Gamma \vdash \text{succ}(n) : \text{nat}}$$

6.6 Inductive-Inductive Types

6.7 Quantifiers

6.8 Girard's Paradox

Addendum

ree