A Primer on Homotopy Type Theory Part 1: The Formal Type Theory

James Ladyman & Stuart Presnell
james.ladyman@bristol.ac.uk
stuart.presnell@bristol.ac.uk

Friday 21st November, 2014

Abstract

This Primer is an introduction to Homotopy Type Theory (HoTT). The original source for the ideas presented here is the "HoTT Book" – *Homotopy Type Theory: Univalent Foundations of Mathematics* published by The Univalent Foundations Program, Institute for Advanced Study, Princeton. In what follows we freely borrow and adapt definitions, arguments and proofs from the HoTT Book throughout without always giving a specific citation.[1] However, whereas that book provides an introduction to the subject that rapidly involves the reader in advanced technical material, the exposition in this Primer is more gently paced for the beginner. We also do more to motivate, justify, and explain some aspects of the theory in greater detail, and we address foundational and philosophical issues that the HoTT Book does not.

In the course of studying HoTT we developed our own approach to interpreting it as a foundation for mathematics that is independent of the homotopy interpretation of the HoTT Book though compatible with it. In particular, we interpret types as concepts; we have a slightly different understanding of subtypes and the Curry-Howard correspondence; and we offer a novel approach to the justification of the elimination rule for identity types in section 7 below (though it builds on a known mathematical

[1] Page numbers for the HoTT book refer to the edition on Google Books, available at `http://books.google.co.uk/books?id=LkDUKMv3yp0C`. Note that the HoTT Book is being continually corrected and updated. The latest edition (last updated on March 6th 2014 at the time of writing this document) is available at `http://homotopytypetheory.org/book/`.

result). These ideas are developed in detail in our papers downloadable from the project website: `http://www.bristol.ac.uk/arts/research/current-projects/homotopy-type-theory/`. While our ideas and views about some important matters differ from those presented in the HoTT Book the theory we present is essentially the same.

Part I below introduces, explains and justifies the basic ideas, language and framework of HoTT including propositional logic, simple types, functions, quantification and identity types. In the subsequent parts of this Primer we extend the theory to predicate logic, the theory of the natural numbers, topology, the real numbers, fibre bundles, calculus and manifolds, and the very important 'Univalence axiom'.[2]

[2] We are very grateful to Steve Awodey for feedback on our work on identity types, and to John Wigglesworth for comments on a draft of this material, and also to Richard Pettigrew and participants in our seminar on HoTT at the University of Bristol. We are immensely grateful to the Leverhulme Trust for funding the project of which this Primer forms part.

Contents

Part I

The Formal Type Theory

Summary

In Part I we introduce the basic elements of the language of Homotopy Type Theory. Section 1 motivates the idea of type theory as a foundation for mathematics, and in particular *constructive intensional* type theory. We also introduce the 'Propositions as Types' approach, which allows a constructive logic to be incorporated into the basic structure of the type theory itself, rather than having to be provided independently.

In Section 2 we start to build up the type theory used in HoTT by introducing the method of defining *functions*, namely *lambda abstraction* from lambda calculus. No previous familiarity with lambda calculus is assumed as we develop the ideas from basic principles.

In Section 3 we develop the Propositions as Types approach in more detail, explaining how the basic logical operations of conjunction, disjunction, implication and negation should be understood on this view. This leads to the Brouwer-Heyting-Kolmogorov (BHK) interpretation of *constructive* (or 'intuitionistic') logic, which is the basis of HoTT's incorporation of logic into type theory.

In Section 4 we define the types that implement the logical connectives discussed in the previous section: the *product* type that plays the role of conjunction, the *coproduct* type that plays the role of disjunction, and the *Zero* type that plays the role of an 'impossible' or 'contradictory' proposition, allowing negation to be defined.

In Section 5 we take a break from the development of the formal language to investigate the system built in the previous sections. We show how the basic theorems of propositional logic are stated and proved in HoTT, and how the constructive logic that follows from the BHK interpretation does not endorse the laws of Excluded Middle and Double-Negation Elimination, and in this way differs from classical logic.

In Section 6 we add further elements to the language in order to be able to do existential ($\exists$) and universal ($\forall$) quantification. This requires the introduction of *predicates*, which in turn requires that a *higher-order* type is introduced. This appears at first to be dangerously similar to the idea of a 'set of all sets' that is at the heart of Russell's Paradox. Section 6.1 shows how this paradox is avoided.

Section 7 introduces the final piece of the language that allows us to express *equality*. This addition to the language requires us to re-think (Section 7.3) our earlier assumptions about the elements we have previously introduced. Then, after reflecting on the meaning of identity in mathematics, we obtain a powerful rule for proving theorems about identifications called *based path induction*.

Univalence and Applications

Part I presents the language of HoTT but (aside from propositional logic in Section 5) does not show how it can be used to do mathematics. Examples of applications, such as defining the natural numbers and recursive functions on them, are given in Part II.

We also do not introduce the important ideas of *function extensionality* (an identity criterion for functions) and *Univalence* (an identity criterion for types). These play a vital role in the theory and applications of HoTT, and Univalence is arguably the most important contribution of HoTT to mathematics. However, they are not required in order to understand the basic theory, and are best presented as part of a more detailed discussion of the general notions of *identity* and *equivalence*. We therefore defer these ideas to a later part of the Primer.

Homotopy and Foundations

HoTT is closely connected with a branch of existing mathematics called 'homotopy theory' and the relationship between them provides many important tools and techniques that are vital for its application. The presentation in the HoTT Book emphasises this aspect, introducing the connection with homotopy theory from the start and relying upon it to motivate and justify various elements of the theory (in particular the principle of 'path induction', discussed here in Section 7).

While such a presentation is very helpful for readers already familiar with the ideas of homotopy theory, the presentation in this Primer is intended for a broader audience. The basic ideas of HoTT can be developed without drawing upon this homotopy interpretation, and so in Part I we explain and justify the basic elements of HoTT without recourse to any sophisticated prior mathematics, introducing these ideas in a later part of the Primer as an optional way of interpreting the type theory. Presenting the theory in this way, using only elementary pre-mathematical ideas, also supports the foundational claims of Homotopy Type Theory: that it may be a starting point upon which all of mathematics can be built (see Ladyman

and Presnell 'Does HoTT Provide a Foundation for Mathematics?').

1 Type Theory

1.1 Motivation

In mathematics we define and study objects and structures of various kinds, such as integers, real numbers, vectors, groups, manifolds, and fibre bundles. Different kinds of things can be treated and manipulated in different ways: we can take the square root of a real number or try to construct a connection on a fibre bundle, but it makes no sense to try to take the square root of a fibre bundle or put a connection on a real number. This idea that there are different kinds of mathematical entities, and operations that can be performed with some kinds but not others, goes back to antiquity: Euclid in the *Elements* distinguishes points and lines in the plane as two distinct kinds of things, and, for example, defines the intersection of two lines but not the intersection of two points, since the latter makes no intuitive sense.[3]

A **type theory** or **type system** is a formalisation of this idea. Whereas in ordinary informal mathematical practice we are guided by an intuitive understanding of the distinction between different kinds of mathematical entities, in a type theory these distinctions are made formal, and enforced and studied.

Ancient mathematics was concerned with concrete operations that could be performed in physical space, such as the bisection of a line, or with physical objects, such as the addition of two collections of units. For these purposes a purely intuitive conception of the different kinds of mathematical entities under discussion was sufficient, since they were directly related to concrete things.

However, the history of mathematics has been one of increasing abstraction. For example, considerations of the ways in which a set of objects can be rearranged gave way to the study of permutation groups, and then to the abstract study of group theory, which allowed the definition of groups of continuous symmetries and their infinitesimal generators.

We are consequently led to domains of study that are far removed from direct human experience. Many mathematical structures have only the remotest connection to the physical universe, if any; in so far as they are accessed it is by thought and language.

Hence, in order to be sure they are operating with a common conception of their subject matter, mathematicians cannot rely on their individual intuitions and must give rigorous definitions of the entities they are studying and the permissible operations that can be performed on them. As the domain of study moves further from

[3] See Kamareddine, Laan & Nederpelt, "A Modern Perspective on Type Theory", Chapter 1 for more discussion of the 'prehistory' of type theory.

direct experience, and as the complexity of the subject increases, the importance of clearly stated definitions correspondingly increases. Many recent proofs (such as Hales' proof of Kepler's conjecture[4]) are so complicated that they can only be adequately verified by making them completely formal so that a computer can examine them.

Homotopy Type Theory (HoTT) is a proposed new foundation for mathematics, grounded in a formal type theory. That is, it provides a formal language in which mathematical definitions and proofs can be built up systematically through basic operations, and in which the idea that different mathematical entities may be of different types is a central organising principle. In the remainder of this section we outline some of its basic features.

Type theory may be used to represent and study logic. The 'Curry-Howard correspondence' is a strong analogy between various logical systems and certain type systems. In particular, a proof in a logical system can be thought of as a mapping from premises to conclusions, analogous to a function that takes tokens of its input type to tokens of its output type.

Logic	proof	premises	conclusion
Type theory	function	input type	output type

This correspondence extends to other features of logic, including logical connectives such as conjunction, disjunction, and negation, and the quantifiers. As explained below, this allows the definition of a type theory in which, alongside the types corresponding to familiar kinds of mathematical objects such as the natural numbers, we also have types corresponding naturally to *propositions* about these entities. In such a system a single set of operations play two roles at once: they are simultaneously mathematical constructions like *forming pairs of elements* and also logical operations such as *conjunction*. Likewise, the basic rule of logical deduction, *modus ponens*, follows from the general rule for applying functions to arguments. This makes it possible to construct a unified language in which logical and mathematical proofs can be expressed, and in which – since the language is grounded in a formal type theory – the correctness of proofs can be verified automatically by a computer. That is, HoTT provides a *programming language for mathematics*.

There are a variety of automated proof verification systems such as Automath, Metamath, HOL, Isabelle, and Mizar.

However, according to Vladimir Voevodsky, one of the founders of Homotopy Type Theory, progress using these is often impeded by problems that stem from their

[4] See `code.google.com/p/flyspeck/`

use of the standard set theoretic foundation for mathematics.[5] The alternative foundation provided by HoTT avoids these problems by design (as we explain in Section 1.3) and can therefore be readily used to formalise a much wider domain of mathematics. In principle, all of modern mathematics can be expressed in this language, and thus all proofs can be checked and verified automatically.

Aside from these practical benefits, a foundation for mathematics based in type theory has other advantages. It forces us to reassess our understanding of mathematical activity: the logic that naturally emerges from this unification process is not familiar classical logic, but rather a form of **constructive logic**. That is, principles like the 'Law of Excluded Middle' (i.e. that for all propositions P, we must admit either the proposition itself or its negation $\neg P$) and the 'Principle of Double-Negation Elimination' (i.e. that $\neg\neg P$ implies P) are not available as universally applicable laws of logic. However, particular instances of these principles can be assumed whenever they are required, and so classical mathematics can nonetheless be recovered in full. This distinction allows for much finer control over the mathematical machinery that we employ in any particular proof.

The use of constructive logic also enables us to give a firmer philosophical basis for the resulting foundation for mathematics, and eliminates many of the long-standing problems in understanding and justifying mathematical activity – that is, explaining and ensuring that mathematics is not merely playing with words. One approach to this problem is Formalism, the view that mathematics is fundamentally about symbolic systems. Another approach is Constructivism or Intuitionism, which insists that the entities that are discussed in mathematical discourse are somehow 'constructed' from more basic entities. Brouwer and other Intuitionists based such constructions in the mind of the mathematician, and involved mysterious ideas that have proven hard to make clear and coherent. Other forms of constructivism sometimes involve philosophical claims about mathematics that may be unpalatable to the working mathematician. As we argue in Section 1.7, HoTT combines the benefits of Constructivism and Formalism: the constructions in question can be understood purely formally as manipulations of symbols, and no strong position on questions about the ontology of mathematics is required.

Furthermore, this constructive logic is of interest in its own right. It observes distinctions that collapse in classical logic, thereby revealing mathematical concepts that would normally go undiscovered. It also has connections to other areas of study such as computer science, category theory, topology, and homotopy theory. This allows ideas to be taken back and forth between these various domains, inspiring innovations in one area that are the analogues of ideas from another.

[5] See, for example, his talk "Foundations of Mathematics and Homotopy Theory", 2006, available at `http://video.ias.edu/node/68`

For example, the 'Univalence Axiom', a central idea in HoTT, is a new proposed principle of logic that is the analogue of an idea from homotopy theory.

1.2 Basic Ideas of Type Theory

The essential ideas of type theory are:

- A type theory is a formal system consisting of **tokens** and **types**.
- Every token belongs to some unique type; we write `a : A` to denote that token `a` is of type `A`.
- The operations that may be done to tokens are restricted according to their types.
- Types can themselves be tokens of higher order types.

The basic idea goes back to Bertrand Russell's work at the beginning of the 20th century, though it is implicit in Frege's logic. Since an understanding of its origins and history is not necessary in order to gain a sufficient understanding of the subject we do not go into further detail here.[6]

Fundamentally, a type theory is a formal way of dividing up some universe of discourse into different types so that each thing belongs to exactly one type. Different type theories may concern themselves with different universes of discourse, and they may divide things along different lines according to different criteria. The universe of discourse is usually some part of mathematics. However, type systems are also used in computer programming languages, where the entities under consideration may also include, e.g., strings of characters from some alphabet.

The rules of a type theory say what entities it talks about, and into what types it can classify them, as well as what operations can be performed on the tokens. By careful choice of the rules it is possible to define a type theory that encompasses *all* of mathematics – i.e. that is able to talk about any mathematical thing that can be defined – and that classifies them along lines that make intuitive sense to a mathematician while providing powerful tools for carrying out mathematical reasoning. Homotopy Type Theory is exactly such a type theory and this Primer begins to explore the view of logic and mathematics that arises from it.

Some type systems are, roughly speaking, 'finer' or 'coarser' than others, in the sense that a finer system draws distinctions between things that a coarser system

[6] For more details see `http://plato.stanford.edu/entries/type-theory/`

lumps together into the same kind. In this sense, Russell's type theory is rather coarse, distinguishing only between *individuals*, *propositions*, and relations over these. Similarly, Gödel had a type theory that had *individuals*, *classes of individuals*, *classes of classes of individuals*, and so on. To capture fully the intuitive idea of 'different kinds of things' from ordinary mathematical practice a much finer type system than this is required.

Type systems used in programming languages are generally quite fine, in that they typically distinguish between integers, real numbers, boolean variables, and many other types. Further, for any two types `A` and `B` we typically have the **function type** `A → B` of functions whose input must be of type `A` and whose output will always be of type `B`, thus enforcing the idea that certain operations can only be performed on certain kinds of things.

HoTT is similarly fine grained. In particular, there is, for example, a type of *natural numbers*, and a type of *points in the infinite Euclidean plane*, and for any two types `A` and `B` there is the function type `A → B`, and so on. So, as a useful heurisic, we can think of types as *kinds of thing that a mathematical entity could be*, with the things of a particular type being the tokens of that type.

1.3 Intensional Type Theory

All of what is said above is compatible with taking type theory to be a way of stratifying the universe of sets, tokens being taken to be members of their types. However, in HoTT we do *not* interpret types as sets in this way. That is, when we talk about a 'kind of thing a mathematical entity could be' we do not mean the *set* of all mathematical entities of a certain kind. Types are not just particular sets, and tokens of types are not just elements of sets. Rather, we should think of *types* and *tokens* as the primitives of a different alternative foundation for mathematics. The analogy between *types and tokens* and *sets and elements* is sometimes helpful, but it is only an analogy and taking it literally does not give a correct picture. The fundamental difference between HoTT and set theory is that the former is **intensional**[7] while the latter is **extensional**.

In set theory, particular sets are nothing over and above particular collections of entities. So, for example, the set of prime factors of 6 and the set of prime factors of 144 are the very same set, namely $\{2, 3\}$. Likewise, the set of even divisors of nine and the set of even divisors of eleven are the same set, namely the empty set (i.e. the unique set that has no members). Two distinct descriptions pick out one

[7] In fact **hyperintensional** in the terminology of contemporary metaphysics since intensions are said to be the same if they have the same extensions in all possible worlds but mathematics is necessary so the sets above have the same extension in all possible worlds.

and the same set if their extensions (i.e. the collection of entities falling under that description) are the same – this is the 'axiom of extensionality'.

On the other hand, intensions can differ even when extensions are the same. Whereas the extension of an expression is associated with the collection of entities to which the expression refers, the intension may be associated with the *sense* or *meaning* of an expression. For example, the intensions of 'featherless biped' and 'rational animal' are different but the extension of each expression is the same (namely the collection of human beings).

Types in HoTT are characterised intensionally – i.e. by how they are described – rather than by what belongs to them. This decision has consequences, as explained below, that may seem unusual, so we should give some justification for it.

Recall that part of the aim here is to provide a 'programming language for mathematics' – a system that is rich enough to encompass all of mathematical practice, but formal so that the correctness of proofs can be verified entirely automatically by computer. The basic notions that we work with must therefore be presented in an entirely clear manner. In particular, any equivalences between entities that are endorsed by the system should be verifiable just by computational means. But things defined extensionally, such as the sets of standard set theory, do not in general satisfy this criterion: we may give two definitions that in fact pick out the same extensional collection of entities, but this fact cannot be verified by any known algorithm.

For example, consider the descriptions 'positive natural number less than 3' and 'natural number n for which the Fermat equation $x^n + y^n = z^n$ has at least one solution in the positive integers'. As Andrew Wiles proved, these two descriptions pick out the same set of numbers. But until his proof was completed we had no way of verifying this fact, and so could not know whether the two described sets were equal or distinct.

In a system in which entities are distinguished extensionally, substituting one description for another that picks out the same extensional collection should leave everything unchanged. But in any system that we could actually implement, verifiable proofs that used one description for a particular entity could become unverifiable when we substitute an equivalent description, since we might be unable to confirm that we're talking about the same things. In practice we are limited by what we can prove, rather than simply what is mathematically true, and any foundational system that is intended to be useful in automated proof verification must recognise this and take account of it from the start.

Founding mathematics in an intensional type system allows us to make the equivalence of any entities *manifest* and so immediately determinable. Any question

of the equivalence of types is either immediately determined simply by comparing the descriptions themselves[8] or by exhibiting an explicit proof of their equivalence within the language of HoTT. There are no merely implicit (and perhaps unprovable) equivalences such as may exist in an extensional system, which hinder and obstruct automatic proof verification.

A consequence of intensionality is that there can be multiple types having *no tokens at all*. For example, *even prime number larger than 2* is a type, and *even divisor of 9* is a type – but there are no tokens of either type. However, despite the fact that both types are extensionally empty, they are different types.

We said above that, as a useful heuristic, types can be thought of as 'kinds of thing a mathematical entity could be'. However, it's a necessary fact that there are no even divisors of 9, and so it's impossible that there be any, and so no mathematical entity could be one. This illustrates the limitations of the above heuristic.

A better way of understanding types that accords very well with their intensional nature is to think of types as *concepts*. It is a straightforward fact of mathematical practice that every well-formulated definition we give in mathematics is the careful expression of some concept, usually formed by putting together pieces corresponding to simpler concepts. Moreover, it can be of great mathematical significance to learn that two mathematical concepts have the same extension. This all points to the fact that it is the intensional things – the concepts and definitions – that we get our hands on most directly in mathematical practice, while the extensions follow afterwards. It therefore makes sense to build the foundations of mathematics in an intensional type system that can track these differences, rather than an extensional one that necessarily discards them.[9]

Finally: what types are there? For now, we have no reason to put any kind of limits on what types there are, so any 'kind of mathematical entity' corresponds to a type, where as usual in mathematics genuine kinds of mathematical entities are those that are well-defined. In HoTT well-defined means that more complex types can be constructed from simpler ones according to a system of rules (to be explained in what follows), and there are certain canonical simple types that are introduced as the primitive types of the theory.[10]

[8] Up to small modifications: for example, if we had called the three variables in the Fermat equation a, b, and c instead of x, y and z, this change would not result in a different type. The details of exactly what modifications are allowed need not concern us at the moment.

[9] Note that this understanding of types as corresponding to mathematical concepts does not commit us to a Brouwerian account of 'mental constructions'. See Section 1.7 for further discussion.

[10] If you are now asking "What about *type* – is that a kind of mathematical entity? Is there a type of all types?" then feel free to look ahead to Section 6, where we address this issue. If you weren't asking that question, feel free to ignore it until we get to Section 6, since it won't bother

1.4 Subtypes

In (extensional) set theory we define a subset S of a set A to be a set containing only some but not necessarily all of the elements of A; we correspondingly call A a superset of S. Thus any element of S is also an element of A. For example, the odd numbers are a subset of the natural numbers, so the odd number 3 is also a member of the set of natural numbers.

However, we have said that in type theory each token belongs to exactly one type. How, then, can we represent the fact that the number 3 belongs to both the odd numbers and the natural numbers (and the primes, and various other subsets of the naturals, of course)? In this section we briefly sketch the solution to this problem, in order to give a clearer picture of what the tokens of types are like.

The basic idea is to use something corresponding to the axiom scheme of Separation in ZFC set theory that defines sets of the form:

$$\{x \in A \,|\, P(x)\}$$

where A is any set and P is any predicate we can define in our language. Anything defined in this way is evidently a subset of A, and there is an obvious injective function from it to A. There is a similar construction in the language of HoTT that defines 'subtypes' of a given type, which can be read analogously. A token of the subtype will be a *pair* consisting of a token $\mathtt{x}$ of $\mathtt{A}$ along with another token that serves as a 'certificate' (see Section 1.5 below) that $\mathtt{x}$ satisfies predicate $\mathtt{P}$.[11]

So, for example, by defining predicates corresponding to 'odd' and 'prime' we can produce a certificate to to the fact that some number $\mathtt{x}$ is odd (which we might denote $\mathtt{odd^x}$), or that some number $\mathtt{y}$ is prime (which we might denote $\mathtt{prime^y}$). Then the pair $(\mathtt{x}, \mathtt{odd^x})$ is a token of the type *odd numbers*, while $(\mathtt{y}, \mathtt{prime^y})$ is a token of the type *prime numbers*. Then, to take the intersection of two subsets we simply combine certificates, so that, for example, a token of the type *odd primes* would take the form $(\mathtt{z}, (\mathtt{odd^z}, \mathtt{prime^z}))$.

Thus while the tokens of the subtype are not literally the same as those of the supertype, the relation between them is obvious – to recover the token $\mathtt{x}$ of the supertype from a token $(\mathtt{x}, \mathtt{odd^x})$ of the subtype, we simply discard the certificate (i.e. project out the first element of the pair). This gives a straightforward 'injection' function just as we have in set theory and allows to represent the facts mentioned above.

us until then.

[11] The type theoretic treatment of predicates is explained in Section 6 below.

This approach neatly combines the following three principles without ending up with a profusion of 'clones' of familiar mathematical entities.

(i) any 'kind of mathematical entity' or mathematical concept corresponds to a type;

(ii) types are distinguished by their descriptions;

(iii) each token belongs to exactly one type.

Even though there are distinct types *natural numbers*, *odd numbers*, *prime numbers*, and so on, we do not face the problem of artificially and arbitrarily distinguished copies of, say, the number 3 in each type – for example, *3 as a natural number*, *3 as an odd number*, and *3 as a prime number*. The use of 'certificates' sketched above (which is the subject of Section 1.5) allows us to construct tokens of all of these types that are distinct from each other in a natural and obvious way and related through predicates and subtypes.[12]

1.5 The Curry-Howard Correspondence and Propositions as Types

As mentioned in Section 1.1, type theory can be used to represent and implement logical reasoning. It was noted by Curry and Howard (and also by Brouwer, Heyting, and Kolmogorov) that there is a correspondence between type theory and natural deduction. Operations in type theory that involve the construction of an output token from some input tokens (computations) correspond to inferences in the constructive fragment of natural deduction. For example, if we have a function `f` of type $\mathtt{A} \to \mathtt{B}$ and a token `x` of type `A` then applying `f` to `x` gives, by definition, a token of type `B`. This is formally parallel to the rule of *modus ponens*: if proposition $A \Rightarrow B$ is true, and proposition A is true, then it follows that proposition B is also true. The Curry-Howard correspondence (or 'equivalence' or 'isomorphism') extends this to other logical operations. Some of its relevant details are explored below.

Note that in the above example the propositions A, B, and $A \Rightarrow B$ correspond to types `A`, `B`, and $\mathtt{A} \to \mathtt{B}$. This will be the case in general: to each *proposition* (which is given by a formula in natural deduction) we associate a corresponding *type*, and a given proposition being *true* corresponds to the appropriate type being **inhabited**

[12] One way formally to define subtypes is in terms of **dependent pair types** as explained in Section 6.3.2. In Part II of this Primer we present another way and go into more detail about subtypes.

in the sense that we can produce a token of that type. For example, propositions like '7 is prime' and 'every complemented distributive lattice is a Boolean algebra' correspond to types. (We do not consider propositions about concrete reality such as that snow is white or that water is H_2O.)

However, we've also said that types are, roughly, 'kinds of thing that a mathematical entity could be'. So for a given proposition P, what are the mathematical entities that play the role of the tokens of type `P` – the things whose existence corresponds to the proposition P being true?

In some cases we can immediately see what such a thing might be. For example, consider the proposition that two entities X and Y are isomorphic (i.e. that there exists some isomorphism between them). The tokens of the corresponding type are precisely the isomorphisms between X and Y (if there are any). More generally, if P is the claim that there exists such-and-such kind of a thing (or can be naturally interpreted in this way, as in the case of isomorphism above) then of course we can take the tokens of the corresponding type to be just those things whose existence is asserted.

However, there are many mathematical propositions that are not naturally read as existential claims, for example, 'all divisors of three are also divisors of nine'. What mathematical entity could serve as a token of the corresponding type?

The most obvious mathematical entity whose existence corresponds to the truth of a proposition is a *proof* of that proposition P, and indeed the Curry-Howard correspondence is generally said to relate tokens in type theory to proofs in logic. However, this is not the interpretation we use, since *proofs* are not in one-to-one correspondence with *tokens*.

To see this, recall from Section 1.1 that the logic we obtain via the Curry-Howard correspondence is *constructive*. Unlike in classical logic, in constructive logic a proof of P requires that there be some means of constructing a thing that stands as conclusive evidence for the truth of the proposition. In later sections we spell out the various rules of construction, and see many examples of constructive proofs, which will make these ideas clear. A proof in the formal system of HoTT is a process in which these rules of construction are applied in some sequence. The end result of a proof is a token of the type corresponding to the proposition to be proved, but there can be multiple distinct proofs that result in the same token being constructed. So whereas a proof is a construction process, a token of a type is rather the end result of a proof. If `P` is the type corresponding to proposition P, then we may call the tokens of `P` '**certificates** of P'.[13]

[13] In the HoTT Book (p. 18) these are also called 'witnesses' or 'evidence' for the proposition. We explain our terminology and other related issues in our paper 'Does HTT Provide a Founda-

This allows us to fit the *propositions as types* account into the general model of types as 'kinds of thing a mathematical entity could be' or mathematical concepts. Since we construct certificates to propositions by means of mathematical and logical rules, certificates are mathematical concepts. For each proposition P, then, *certificate to* P is an instance of a mathematical concept or a kind of thing that a mathematical entity can be, and therefore constitutes a type. Just as every proof is a proof of a particular proposition, so each certificate is a certificate to exactly one proposition, and therefore belongs to exactly one type. Certificates can therefore serve as the tokens of these types, and *certificate to* P is the mathematical concept corresponding to proposition P.

1.6 Proof relevance

There is another unfamiliar feature of the Propositions as Types approach to logic that merits attention. In many systems of logic, propositions are regarded as binary entities – they take on exactly one of two 'truth values', namely *true* or *false*. There are, of course, many-valued logics in which we work with more than two truth values, perhaps uncountably many. But even in these systems, once we have determined the truth value of a proposition then the particular way in which we demonstrated that truth value is no longer relevant, and the value itself is all we need to carry out future computations.

In HoTT, on the other hand, a proof of a proposition produces a particular token of the corresponding type, and two different proofs of the proposition may construct two different tokens. Since in general the tokens of a type are not interchangeable, it may be important not only that we can produce some token of the type (and thereby prove the corresponding proposition), but also *which* particular token was produced. This is called a **proof relevant** approach to logic.

Working in a proof relevant style does not, in general, place an extra burden on the mathematician (over and above the use of constructive logic). Indeed, in some cases it is a more natural thing to do and can make our work simpler.

To illustrate this, imagine that we have carried out a non-constructive proof that two structures are isomorphic. To carry some property from one structure to the other requires a particular isomorphism between them and so we may have to invoke the Axiom of Choice to produce one arbitrarily (since we know only that an isomorphism exists but have no way to produce any particular one). If we had instead worked constructively then our original existence proof would necessarily

tion for Mathematics?', available in draft form at `http://www.bristol.ac.uk/arts/research/current-projects/homotopy-type-theory/`.

have constructed a particular isomorphism.

Working in a constructive but proof-*irrelevant* way corresponds to constructing a particular isomorphism but then 'throwing it away' and retaining only the knowledge that some isomorphism exists, which leaves us in no better position than the non-constructive mathematician. Working proof-relevantly, on the other hand, simply corresponds to keeping the particular isomorphism for later use, rather than discarding it. The use of proof-relevant methods therefore mitigates some of the difficulties that may be thought to arise from using constructive logic, and the associated restriction on the use of the Law of Excluded Middle and the Axiom of Choice.[14]

In Part IV we explore the extent to which a more familiar *proof irrelevant* approach can be recovered, thus showing that while a proof relevant style fits naturally with constructive logic, it does not follow automatically or unavoidably from the use of constructive logic.

1.7 Philosophy

We end this introductory section with some remarks on the philosophical background to the project.

The explosion of work in mathematical logic and set theory in the early twentieth century was accompanied by the development of three schools in philosophy of mathematics, namely Platonism, Formalism, and Intuitionism. Platonism is roughly the view that mathematical entities exist independently of our minds. Formalism is roughly the view that mathematics is about the manipulation of symbols in accordance with rules. Intuitionism is the view that mathematical entities are creations of the mind. Intuitionism leads directly to the use of constructive logic,[15] but the use of constructive logic does *not* commit us to an Intuitionist philosophy of mathematics. In particular, it does not require us to believe that mathematical entities do not exist before some mathematician has defined them, nor that mathematical propositions 'gain a truth value' only when some mathematician has proved them.

When it was originally proposed by Brouwer, Intuitionism was intended as a thorough reappraisal of all of the underpinning ideas of mathematics. Traditional foundational ideas about the nature of mathematical entities and proofs were supposed to be overturned. Brouwer, troubled by classical mathematicians' handling

[14] "The axiom of choice is used to extract elements from equivalence classes where they should never have been put in the first place." (Bishop, *Foundations of Constructive Analysis*, p. 9)

[15] See Chapter 1 of Michael Dummett's "Elements of Intuitionism".

of entities that are far from any possible human experience, advocated a position that makes narrower claims about what mathematical entities exist, and ties them more directly to human experience and activity. In particular, the existence of entities is only inferred when they can be directly constructed according to some finite set of rules in accordance with our intuitions. This is to adopt a constructive attitude to the **ontology** of mathematics.

However, concerns about the existence of mathematical entities is not the only possible motivation for using constructive logic. Indeed, we might choose to work with a constructive rather than classical logic while being uninterested or untroubled by ontological issues. We may adopt the position that questions about what mathematical entities *exist* need not trouble us, so long as we suitably restrict ourselves regarding what entities we can claim to *know about*. Constructive logic provides a secure basis for our claims to knowledge about mathematics, since we only talk about things that are known to exist (since we construct them directly), and everything we prove is guaranteed to have a certificate that certifies its truth. This is to adopt a constructive attitude to the **epistemology** of mathematics.

On the other hand, we might say that philosophical worries of this sort are not relevant at all to our work as mathematicians, and yet still prefer to use a constructive logic for practical reasons. This may be because constructive proofs always produce certificates to propositions that can be used in a proof-relevant manner rather than simply establishing mathematical facts as true or false (as discussed in Section 1.6) and are therefore more useful. We may also prefer constructive proofs because they can be easier to verify. We might, therefore, take a constructive attitude to the **methodology** of mathematics.

Any of these reasons – ontology, epistemology, or methodology – can lead us to the use of constructive mathematics without being motivated by, or committing to, Brouwer's Intuitionist philosophy of mathematics.

Moreover, commitment to constructive methodology can come in different degrees. Constructive logic does not incorporate the Law of Excluded Middle (LEM) as a general law of logic; however, this does *not* mean that for each and every proposition P we *reject* the disjunction $P \vee \neg P$. That is, constructive logic does not adopt the denial or negation of LEM.[16] Thus for any given proposition Q we can always introduce $Q \vee \neg Q$ as an *assumption*, and use this in a proof that is otherwise constructive. Constructive logic does not *forbid* us from using instances of $P \vee \neg P$, it simply forces us to take note of any such instances we use and either prove them or acknowledge them as unproven assumptions.

So if, for whatever reason, we want to work in a mathematical framework or

[16] Moreover, the negation of LEM leads to contradiction (see Theorem 20 in Section 5.5).

style that leans toward constructive methods but is not fully constructive, we can achieve this by working in a strictly constructive logic and then introducing non-constructive assumptions as and when they're needed. This is easier than working in a classical framework that includes LEM and then imposing the restrictions of constructive logic later.

In any case, as a subject of mathematical study, constructive logic is of interest independent of any particular philosophical motivations, in particular for its connections to computer science and category theory. Also, constructive logic allows us to study a broad spectrum of new mathematical objects and properties that are either forbidden or obscured classically.[17]

In the present project we set aside ontological questions about mathematics,[18] and adopt constructive logic for epistemological, methodological, and mathematical reasons. Constructive logic provides essential connections between logic and other domains, in particular topos theory and homotopy theory, that are central to the HoTT project. It forces us into methods of proof and definition that are more explicit and therefore more amenable to formalisation in a language that can be used by computer programs to assist in doing mathematics.[19]

[17] For more details on the merits of constructive logic, and its surprising departures from classical logic, see Andrej Bauer's talk "Five Stages of Accepting Constructive Mathematics", available at `http://video.ias.edu/members/1213/0318-AndrejBauer`.

[18] Essentially, we are assuming that the mathematics founded in constructive logic that we study is no worse off as regards ontological questions than any other branch of mathematics on any other foundation. Further, we assume that whatever solutions can be found to philosophical problems regarding ontology of mathematics in other settings can equally well be applied here, in a way that is compatible with the foundational assumptions we've made. HoTT fits particularly well with structuralism in the philosophy of mathematics (see Steve Awody "Structuralism, Invariance and Univalence') but we do not pursue this in the present work.

[19] For more details of our views on philosophical issues related to HoTT see our paper 'Does HoTT Provide a Foundation for Mathematics?', available at `http://www.bristol.ac.uk/arts/research/current-projects/homotopy-type-theory/`.

2 Expressions and Functions

In this section we begin to build the language of HoTT.

Functions are central to type theory. In HoTT functions are an essential tool for defining and using types and tokens. To define any particular type we must specify how to *produce* tokens of that type and how to *use* tokens that we are given. Both these things are done with functions into and out of the type in question (see Section 4 for details). Hence, we need to know how to define and use functions before we can do anything else.

As we mentioned when introducing the Curry-Howard correspondence (Section 1.5), the function type between `A` and `B` is written `A → B`. That is, if expressions '`A`' and '`B`' name types, then the expression '`A → B`' also names a type. Tokens of this type are understood as functions whose input is of type `A` and whose output is of type `B`. But what exactly *are* functions? Colloquially, we often talk about functions very loosely, treating them as 'gadgets' or 'devices' that 'take in' inputs and 'return' outputs. But if we are to use them as a basic part of the foundations of mathematics we need to give them a rigorous definition. (After this we can go back to talking loosely, safe in the knowledge that this can be translated to something more formal).

The following discussion may seem a little heavy-going, because we need to draw attention to the distinction between expressions in the language and the abstract things they name. In subsequent sections we will suppress this, but it is important that the relevant distinction between the linguistic and the non-linguistic is understood. Once we've done this the definitions of other types can be given by using functions, and direct reference to expressions can be left behind, making things much simpler.

2.1 Expressions

We begin by discussing the formal expressions used in HoTT, which correspond to the sequences of symbols that are directly manipulated by computer programs.

Until now we've mainly been talking about mathematical entities in English rather than in a formal language, using phrases like '*natural numbers*' to denote types, and saying things like '7 is a prime number'. But of course we'll want to be able to discuss type theory in a general abstract way without always coming back to specific mathematical examples, and without making any particular assumptions about what types we have defined and what tokens of those types we have. We therefore need a formal language to work in, i.e. a **syntax** that tells us how we can

put elements of the vocabulary together to make expressions, and a **semantics** that tells us how to interpret these expressions.

In what follows we denote arbitrary types by capital letters '$\mathtt{A}$', '$\mathtt{B}$', '$\mathtt{C}$' ... and we generally denote tokens with lower case letters '$\mathtt{x}$', '$\mathtt{y}$', '$\mathtt{z}$' .... The fact that $\mathtt{x}$ is a token of type $\mathtt{A}$ is written as '$\mathtt{x} : \mathtt{A}$'.

Subsequent sections introduce further syntactic elements which allow us to take expressions that name tokens and types and construct from them new expressions that name other tokens and types. For example, if '$\mathtt{A}$' and '$\mathtt{B}$' are the names of types, and '$\mathtt{a}$' and '$\mathtt{b}$' are the names of tokens of these types, then '$\mathtt{A} \times \mathtt{B}$' is the name of another type, and '$(\mathtt{a}, \mathtt{b})$' is the name of a token of this type.

The **semantics** assigning meaning to such expressions is as follows:

(i) Most expressions are the *names* of tokens or types in the type theory; we call these *naming expressions*. Each such expression names exactly one token or type – the evaluation of expressions is always entirely unambiguous. Moreover, any such expression constructed using these rules is the name of a token or type. That is, if we can construct it then it is a valid name. (With one exception: see footnote 20.)

(ii) A given token or type may have multiple different names that express it in the language. For example, when we introduce the natural numbers we will use both a unary representation as a sequence of successor operations, such as '$\mathtt{s}(\mathtt{s}(\mathtt{s}(0_{\mathbb{N}})))$', but we will also use ordinary numerals like '3' for convenience. These two expressions are two names for the same token.

An expression of the form

$$exp_1 :\equiv exp_2$$

(where exp_1 and exp_2 are both names of tokens or both names of types) introduces the expression exp_1 as a new name for whatever token or type is named by exp_2.

(iii) Expressions such as '$\mathtt{x} : \mathtt{A}$' assert that the token named by the expression '$\mathtt{x}$' belongs to the type named by the expression '$\mathtt{A}$'. (We sometimes abuse this notation slightly and say something like 'given a token $\mathtt{x} : \mathtt{A}$ ...', where what we mean is 'given a token $\mathtt{x}$, where $\mathtt{x} : \mathtt{A}$, ...'.)

These are the main kinds of expressions used in HoTT. The following sections (starting at Section 4) introduce and explain the main vocabulary and rules of construction that are used to construct the names of tokens and types.

In sum this is the overall set-up with which we're working:

(a) We have a language consisting of some basic vocabulary and some construction rules, and we use this language to construct names of tokens and types.

(b) Every naming expression we can construct using these rules is the name of a token or a type.

(c) Every token belongs to exactly one type.

(d) Proving a proposition involves constructing an expression that names a token of the corresponding type using the rules of construction. If the proposition is being proved from some premises then the construction may use tokens of types corresponding to the premises as inputs.

Point (b) is what guarantees that successive applications of the construction rules constitute proofs of propositions. We might think that merely constructing an expression could not serve as a proof of the corresponding proposition without an additional *existence proof* demonstrating that there really is something named by this expression. To understand this we must take care to distinguish expressions naming types from those naming tokens of those types.

In HoTT, as in other frameworks, we can construct an expression corresponding to, for example, 'even divisor of 9'. But this names a *type*, not a *token*, and therefore does not conflict with the fact that 9 has no even divisors. Our ability to construct names of types corresponds to our ability to *express* or *state* propositions. But of course merely stating a proposition is not the same as *proving* it. When we prove a proposition we do so by constructing an expression naming a *token* of the corresponding type. The rules of construction are chosen so that point (b) holds, and so constructing the expression guarantees that the named token exists. Thus the type that the token belongs to is inhabited, which certifies that the corresponding proposition is true. In summary: a proof of a proposition constructs an expression that is guaranteed to name a token of the corresponding type, whose existence then certifies the truth of that proposition.[20]

The above picture provides the background, but in practice we generally won't talk explicitly about syntactic matters at all, instead talking about 'constructing a token' and 'defining a type' rather than the literally correct but more verbose 'constructing an expression that names . . . '.

[20] There is an exception to this: if we are given *inconsistent* premises then we may be able to construct from them an expression that does not name any token – i.e. an *empty name*. See Section 3.3 for further discussion, and Section 5 for examples.

2.2 What functions can we define?

Before we work out *how* to define functions, we should consider the question of *what* functions we should expect to be able to define, since this will guide our thinking on what functions are.

In set theory, a (total) function between sets A and B is an arbitrary pairing of elements of A and B, in such a way that each element of A is paired with exactly one element of B. We say that when the function is given a element of A as input it returns the corresponding element of B as output. However, this allows us to define functions that are *uncomputable* in the sense that there is no finite process that can be carried out by a computer that is guaranteed to return the specified output when given a particular input.

Since part of the aim of this project is to define a framework for mathematics that can be implemented in a computer program, we require that every function that can be defined must be a *finitely computable procedure*. That is, it must be specifiable in a finite set of instructions and must be guaranteed to terminate within a finite number of computational steps. Thus a function cannot just be a pairing of arbitrary inputs to arbitrary outputs as in set theory.[21]

We therefore turn to a different style of function definition that ensures the computability of all functions that it can define – specifically, Church's **lambda calculus**.[22] In this system functions are defined via manipulations of expressions: in summary, the application of a function to an argument involves replacing certain parts of an original expression with a new sub-expression (which names the argument), thereby transforming it into a new expression (which names the output). Since this is a simple process of *rewriting*, each such step is computationally trivial to carry out. It can be shown that in this system any sequence of successive allowed re-writings will eventually terminate, as required.[23]

In the remainder of this section we show how this style of function definition works, starting from first principles.

[21] "If a function is defined as a binary relation satisfying the usual existence and unicity conditions, whereby classical reasoning is allowed in the existence proof, or a set of ordered pairs satisfying the corresponding conditions, then a function cannot be the same type of thing as a computer program." Per Martin-Löf, "Constructive Mathematics and Computer Programming" (1984) *Philosophical Transactions of the Royal Society of London A*, **312**, pp. 501–518.

[22] See `http://plato.stanford.edu/entries/lambda-calculus/`

[23] We won't actually prove this here. See Appendix A.4 of the HoTT Book for details.

2.3 Free and Bound Variables

Many naming expressions[24] involve sub-expressions that name other tokens. This is familiar from arithmetic and algebra: the expressions '3' and '5' name integers, '$3+5$' names another, and '$3/5$' and '$(3+5)/5$' both name rational numbers. Thus an expression naming a token of one type can involve sub-expressions naming tokens of the same type or of other types.

In algebra we also use **variables**: symbols in the vocabulary that are stipulated, for the purposes of a calculation, to name some token of some particular type. Whereas expressions such as '$(3+5)/5$' name *particular* tokens, variables do not – we may say that they name 'unknown' tokens of that type whose particular value has not yet been established. Expressions involving variables are then stipulated to name 'unknown' tokens of the corresponding types. For example if we stipulate in a calculation that the symbol 'x' is intended to name an integer ('x' is an 'integer variable' for short), then the expression '$(3+x)$' also names an integer and the expression '$(3+x)/5$' names a rational number (for short, they are respectively an 'integer expression' and a 'rational expression').

Variables are most simply understood as placeholders in an expression, which can be replaced by sub-expressions that name (particular or 'unknown') tokens of the same type. For example, if we take the expression '$(3+x)/5$' and replace the symbol 'x' with an expression naming a particular integer, e.g. '11', then we obtain an expression naming a particular rational number, '$(3+11)/5$'. We could instead replace 'x' with some other integer expression involving a variable, such as '$(2y+7)$', to get a new rational expression '$(3+(2y+7))/5$'. We could also simply replace 'x' with a different integer variable 'z' to get the rational expression '$(3+z)/5$', but this is for all purposes the same.

Expressions can of course contain multiple distinct variables, and multiple instances of a given variable, such as '$(x+2y)/y$'. When we replace a variable with some other sub-expression we must replace *every instance* of that variable in the expression. If the sub-expression we are introducing is a variable, or contains variables, we must ensure that none of these variables are already used elsewhere in the main expression, since such a *collision* (or 'variable capture') may change the intended meaning of the expression.

There is one further distinction that we must pay attention to. In some expressions, such as '$\forall n \in \mathbb{N}, n^2 \geq n$', a symbol such as '$n$' appears that does not name a particular token – so it's a variable – and yet should not be thought of as a placeholder, waiting to have a particular value filled in. We say that 'n' here is a

[24] In the remainder of this discussion we focus only on naming expressions (rather than expressions of the form $exp_1 :\equiv exp_2$ or $\mathtt{x} : \mathtt{A}$).

bound variable – the universal quantifier '$\forall$' *binds* the variable that immediately follows it. Other mathematical operators that bind variables include the existential quantifier '$\exists$', the summation symbol '$\sum$', and the integration symbol '$\int$'. Any variable that is not bound in this way is called a **free variable**.

A bound variable may be replaced with some other symbol, for example replacing 'n' in the above expression with 'k' to get '$\forall k \in \mathbb{N}, k^2 \geq k$'. As with free variables, all instances must be replaced and we must avoid collisions with existing variables in the expression.

Bound variables can only be replaced by other symbols, they cannot be replaced by other expressions. It is only free variables that serve as placeholders in the way described above, and thus only free variables that are candidates for replacement by sub-expressions naming particular tokens or containing other variables.

2.4 Defining functions

This basic idea of replacement of free variables then allows us to define functions. Given an expression Φ of type `B` containing a free variable of type `A` we have a function that we may call $[\mathtt{a} \mapsto \Phi]$.[25] Such a function is a token of type $\mathtt{A} \to \mathtt{B}$. This method of function definition is called **λ-abstraction**.[26]

While the definition of the function is given in terms of manipulation of syntactic elements, we generally suppress this (in accordance with the discussion at the end of Section 2.1) and say that such a function takes a token of type `A` as input and returns a token of type `B` as output. So, for example, given the expression '$(3 + x)/5$' considered above we have the function $[x \mapsto (3 + x)/5] : \mathbb{N} \to \mathbb{Q}$ which takes a natural number as input and returns a rational number as output.

For any types `A` and `B` we have the **function type** $\mathtt{A} \to \mathtt{B}$. That is, if '`A`' and '`B`' are expressions naming types then '$\mathtt{A} \to \mathtt{B}$' is an expression that also names a type. This can of course be iterated: we can form more complicated function types such as $(\mathtt{A} \to \mathtt{B}) \to \mathtt{C}$. A function of this type would take as input a function of type $\mathtt{A} \to \mathtt{B}$ and return as output a token of `C`. Similarly, we may define functions that return other functions as outputs. For example, a function of type $\mathtt{X} \to (\mathtt{Y} \to \mathtt{Z})$ takes as input a token of `X` and returns as output a function of type $\mathtt{Y} \to \mathtt{Z}$.

More generally, wherever in the subsequent notes some operation or construction involves some arbitrary type `A`, this is not restricted only to 'simple' types such as

[25] Strictly, the expression Φ does not need to contain any instances of the variable. If it does not then the function defined will be a constant function (see below).

[26] Traditionally in **lambda calculus**, where this method of function definition originates, functions are written as '$\lambda\mathtt{a}.\Phi$', but we avoid that notation here.

$\mathbb{N}$ and $\mathbb{Q}$, but may just as well be a function type such as $\mathbb{N} \to \mathbb{Q}$ or $\mathtt{X} \to (\mathtt{Y} \to \mathtt{Z})$. Functions are not of a fundamentally different character from any other types in the type theory, and whatever we may do with arbitrary types we may also do with functions – functions are 'first class citizens'.[27]

A particularly simple function is the *identity* function. For any given type $\mathtt{B}$ the identity function is given by a single free variable of type $\mathtt{B}$, and may therefore be written as $[\mathtt{b} \mapsto \mathtt{b}]$ (of course this is a token of $\mathtt{B} \to \mathtt{B}$).

As noted in footnote 25, if the the expression Φ does not contain any instances of the variable $\mathtt{a}$ then the function $[\mathtt{a} \mapsto \Phi]$ is a *constant* function. Whatever value of type $\mathtt{A}$ is given as input to the function, this cannot affect the output value, and so the output is always simply the token of $\mathtt{B}$ named by Φ. As we will see (for example in Section 4.1.3) even trivial-seeming functions such as these can be very important.

2.5 Applying and composing functions

Given a function $[\mathtt{a} \mapsto \Phi]$ of type $\mathtt{A} \to \mathtt{B}$ and a token $\mathtt{x} : \mathtt{A}$, the application of $[\mathtt{a} \mapsto \Phi]$ to $\mathtt{x}$ is written as

$$[\mathtt{a} \mapsto \Phi](\mathtt{x})$$

By definition, the expression '$[\mathtt{a} \mapsto \Phi](\mathtt{x})$' is the name of a token of $\mathtt{B}$, but in general we want to find a better and more useful name for this token. We do this, of course, by the substitution process given above: we replace each instance of $\mathtt{a}$ in Φ by $\mathtt{x}$. This is the 'computation rule' for function types, called β-conversion.[28] For example, the application of the function $[\mathtt{x} \mapsto \mathtt{x} + \mathtt{x}]$ to the token $\mathtt{2}$ is written as $[\mathtt{x} \mapsto \mathtt{x} + \mathtt{x}](\mathtt{2})$, and evaluates to $\mathtt{2} + \mathtt{2}$.

The role of '$\mathtt{a}$' in '$[\mathtt{a} \mapsto \Phi]$' is simply to stand in for the sub-expression that is to replace it. Therefore '$\mathtt{a}$' is no longer a free variable – it has been bound.[29] We can therefore replace '$\mathtt{a}$' with a different symbol, so long as we replace every instance of '$\mathtt{a}$' and avoid collisions (as discussed above). The procedure of consistent non-colliding renaming is called **α-conversion**. Thus, for example, the expressions '$[\mathtt{y} \mapsto \mathtt{x}+\mathtt{y}]$' and '$[\mathtt{z} \mapsto \mathtt{x}+\mathtt{z}]$' are two names for the same function, but '$[\mathtt{x} \mapsto \mathtt{x}+\mathtt{x}]$' names a different function.

Given functions $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ and $\mathtt{g} : \mathtt{B} \to \mathtt{C}$ we can *compose* them to give a function of

[27] Christopher Strachey (2000) "Fundamental Concepts in Programming Languages", *Higher-Order and Symbolic Computation*, 13, pp. 11–49.

[28] The expression obtained in this way is traditionally referred to as '$\Phi[\mathtt{x}/\mathtt{a}]$'.

[29] In the traditional notation, '$\lambda\mathtt{a}.\Phi$', we would say that 'λ' is another operator that binds the variable immediately following it, like '$\forall$' and '$\exists$'.

type $\mathtt{A} \to \mathtt{C}$. We write the composition of $\mathtt{f}$ and $\mathtt{g}$ as $\mathtt{g} \circ \mathtt{f}$ (read as '$\mathtt{g}$ following $\mathtt{f}$'). The effect of applying this function to some input $\mathtt{x} : \mathtt{A}$ is the same as applying $\mathtt{f}$ to that input and then applying $\mathtt{g}$ to the output of $\mathtt{f}$. That is, we define

$$\mathtt{g} \circ \mathtt{f} :\equiv [\mathtt{x} \mapsto \mathtt{g}(\mathtt{f}(\mathtt{x}))]$$

Thus, of course, we can only compose functions $\mathtt{f}$ and $\mathtt{g}$ when the output type of $\mathtt{f}$ is the same as the input type of $\mathtt{g}$. For example, we can compose $[x \mapsto x+2] : \mathbb{N} \to \mathbb{N}$ with $[y \mapsto 3y/5] : \mathbb{N} \to \mathbb{Q}$ to give $[x \mapsto 3(x+2)/5] : \mathbb{N} \to \mathbb{Q}$, but we could not compose the same two functions in the other order.

2.6 Currying: Functions of multiple arguments

The above definition tells us how to form functions that take single inputs, but we also want to be able to define functions with multiple inputs: for example, the function that takes two integers and returns their sum. We can write the expression '$x+y$' with two free variables, but how do we make this into a function?

One obvious solution is that application of a function to multiple arguments is given by *simultaneously* replacing each variable with an expression of the appropriate type.[30] However, the problem with this is that it's not clear what the input type of such a function could be: since all the inputs are treated symmetrically, it seems that the input type ought to be a multi-set of the types of each of the inputs.

Alternatively, rather than replacing the variables simultaneously, we can replace them one by one in some arbitrary order. So for example, to apply the sum function to inputs 2 and 5 we first replace one free variable x with 2 to get the expression '$2 + y$', and then replace the other free variable y with 5 to get '$2 + 5$'.

The final outcome is of course the same as if we'd replaced all the variables simultaneously, but this scheme has the advantage that it already fits into the picture of function definition and function application given above:

- Each step in the process is a function application, i.e. the replacement of one free variable in an expression by an expression of the appropriate type;
- The intermediate expression we produce is again just an expression with a free variable, which therefore gives us a function.

So we can think of the intermediate steps as applications of functions which give us *new functions* as their outputs. In the above example we have one function given

[30] Taking care to avoid collisions, as in α-conversion.

by the expression '$x + y$' and the free variable 'x'; when we apply this function to 2 we get as output another function, defined by the expression '$2 + y$' and the free variable 'y'; we then apply this function to 5 to get the final expression '$2 + 5$', which names an integer. Since we write the latter function as $[y \mapsto 2 + y]$, we must write the original function (which returns this when applied to 2) as

$$[x \mapsto [y \mapsto x + y]]$$

Similarly, since the type of $[y \mapsto 2 + y]$ is $(\mathbb{N} \to \mathbb{N})$, the type of $[x \mapsto [y \mapsto 2 + y]]$ is $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$.

This technique of defining functions of multiple arguments by way of functions that return other functions as their outputs is called 'Currying' (named for logician Haskell Curry).

This straightforwardly extends to functions of more than two inputs, of course. For example, if we want a function whose output of type `Z` depends upon three inputs of types `P`, `Q`, and `R`, then we must define a function of type

$$\mathtt{h} : \mathtt{P} \to (\mathtt{Q} \to (\mathtt{R} \to \mathtt{Z}))$$

If Ψ is the expression of type `Z` that defines this function, containing variables '`p`', '`q`', and '`r`' of the three respective input types, then we might write `h` as

$$\mathtt{h} :\equiv [\mathtt{p} \mapsto [\mathtt{q} \mapsto [\mathtt{r} \mapsto \Psi]]]$$

Note that sometimes the intermediate functions we get are interesting in their own right: for example, if we have a function for raising one number to the power of another, $[b \mapsto [a \mapsto a^b]]$, then applying it to the single input 2 gives us the function that squares its input, $[a \mapsto a^2]$. This method of **partial application** can be a useful way of defining functions, allowing us to move from more general functions to more specialised ones.

Note also that we can only apply the function to its arguments in the order specified: only the variable at the very front of the function expression is available for substitution. Thus we could not, for example, have taken $[b \mapsto [a \mapsto a^b]]$ and substituted 2 for 'a' to give a function $[b \mapsto 2^b]$, since 'a' is not a free variable – it is bound in the function $[a \mapsto a^b]$. However, since we chose an *arbitrary* order in which to bind the variables of the original expression we can just as well take the same expression and choose another order of binding, thereby defining a different function, $[a \mapsto [b \mapsto a^b]]$. Whereas $[b \mapsto [a \mapsto a^b]](2)$ evaluates to $[a \mapsto a^2]$, applying this new function $[a \mapsto [b \mapsto a^b]]$ to 2 gives the function $[b \mapsto 2^b]$.

2.7 Notation

Rather than referring to functions by names of the form '$[\mathtt{x} \mapsto \Phi]$', we generally give them more convenient names like '$\mathtt{f}$' and '$\mathtt{g}$'. We use the '$:\equiv$' notation mentioned in Section 2.1 to introduce new names for tokens:

$$\mathtt{f} :\equiv [\mathtt{x} \mapsto \Phi]$$

The result of applying function $\mathtt{f}$ to $\mathtt{x}$ is written '$\mathtt{f(x)}$', as usual.

It is common in mathematics writing to see the same notation '$f(x)$' used to denote both

(a) a function f that takes a single input, here illustrated with the dummy variable x; and

(b) the value obtained by *applying* such a function to a particular input x.

This notation is ambiguous: the former is a function of some type $\mathtt{X} \to \mathtt{Y}$, while the latter is a value in that function's output type $\mathtt{Y}$. In these notes we will avoid that ambiguity, writing $\mathtt{f}$ for the function itself and reserving $\mathtt{f(x)}$ for the result of applying the function to an input. However, when we're *defining* a function it will usually be more convenient to write this as

$$\mathtt{f(x)} :\equiv \Phi$$

rather than

$$\mathtt{f} :\equiv [\mathtt{x} \mapsto \Phi]$$

This is purely a notational convention, and the former can be thought of as simply a synonym for the latter.

It is sometimes convenient to put the argument of a function as a subscript to the function rather than in parentheses, e.g. writing '$\mathtt{f_x}$' instead of '$\mathtt{f(x)}$'. In particular, we might use this when we have a curried function of multiple arguments and want to write down one of the intermediate functions given by partial application. For example, given the function $\mathtt{h} :\equiv [\mathtt{p} \mapsto [\mathtt{q} \mapsto [\mathtt{r} \mapsto \Psi]]]$ of the previous section, we could provide inputs $\mathtt{p}$ and $\mathtt{q}$ to give a function $\mathtt{h(p)(q)}$ of type $\mathtt{R} \to \mathtt{Z}$. If we want to use this function frequently we might re-name it $\mathtt{h_{p,q}}$.

For functions of multiple arguments it is common to write '$\mathtt{f(x,y)}$' for '$\mathtt{f(x)(y)}$', and we will generally follow this convenient convention to avoid excess parentheses. However, some authors drop the parentheses entirely, simply writing '$\mathtt{f\,x\,y}$' for '$\mathtt{f(x)(y)}$' and relying on the reader to remember the types of all the tokens involved to render this unambiguous. We avoid this notational minimalism in these notes, but this parenthesis-free notation is sometimes used in the HoTT Book (Section 1.2).

3 Logic

As noted in Section 1.5, the Curry-Howard correspondence allows logical operations to be incorporated into type theory by defining suitable types to represent compound propositions. Recall that for each proposition P there is a corresponding type `P` whose tokens are certificates to that proposition. We have seen the correspondence between the implication $A \Rightarrow B$ and the function type $\texttt{A} \to \texttt{B}$, and thus between the rule of *modus ponens* and function application. Now that we have seen how functions are defined we must look at this correspondence more closely.

In this section we will examine how to extend this correspondence to the other basic operations of logic, namely conjunction, disjunction, and negation. These will be sufficient to carry out proofs in *propositional logic*. (In Section 6 we extend the correspondence further to introduce the universal and existential quantifiers $\forall$ and $\exists$ as well, thus allowing us to incorporate *predicate logic* into the type theory.)

These logical operations produce *compound* propositions from *component* propositions. By considering the intended meaning of the compound proposition, and how it relates to the meanings of the components, we can work out how to construct a certificate to each compound proposition from certificates to its components. This in turn tells us how to define the types corresponding to compound propositions, which we do in Section 4. In this section we consider each of the logical connectives in turn, to see what a certificate to each compound proposition should be.[31] In many cases this results in a logical operation that differs from its classical counterpart – we instead get the *constructive* version of the operation.

3.1 Conjunction

What is a certificate to the conjunction $A \,\&\, B$? Classically, $A \,\&\, B$ is true just in case A and B are both true. Since, in the type theory, truth of a proposition corresponds to the relevant types having certificates, the natural candidate for 'certificate to the conjunction $A \,\&\, B$' is a pair of certificates `a` and `b`, where `a` is a certificate to the proposition A and `b` is a certificate to the proposition B. We then find that this behaves like classical conjunction in that from the conjunction we can derive each of the conjuncts, and we can derive the conjunction itself if we can derive both of the conjuncts.

Thus in order for the logic we incorporate to have an operation of conjunction, we

[31] For a more thorough argument along these lines, see Per Martin-Löf, 1996, "On the Meanings of the Logical Constants and the Justifications of the Logical Laws".

need the type theory to have a way of forming (ordered) pairs. Put another way, we need it to recognise that if there can be things of type `A` and things of type `B`, then another kind of thing that a mathematical entity can be is *a pair of things, one of which is of type `A` and the other of which is of type `B`*. Syntactically, we need an operation that takes two expressions that name types and combines them to make an expression naming a new type. We must therefore define the 'pair type' or 'product type', $\mathtt{A} \times \mathtt{B}$. This definition is given in Section 4.1.

3.2 Disjunction

What is a certificate to the disjunction $A \vee B$? Any certificate of A would be sufficient, as would any certificate of B. Thus it would appear that the type corresponding to $A \vee B$ should be the *union* of the two types `A` and `B` – that is, a type that gathers together all the tokens of types `A` and `B`.

But we can't construct a type-theoretic union because, as we noted in Section 1.2, each token has a single unique type and so a token of `A` cannot also be a token of such a 'union type'. The tokens of the type corresponding to $A \vee B$ therefore cannot literally be the same as the tokens of either `A` or `B`.

A similar problem arose in Section 1.4 where we wanted to construct something like a subset, but couldn't have the same tokens belonging to both the subset and the superset. In that case the problem was solved by having the tokens of the subset be suitable *counterparts* of the tokens of the superset, formed by adding a certificate alongside the original token. We take a similar approach here, making '$\mathtt{a}$ as a certificate to $A \vee B$' distinct from '$\mathtt{a}$ as a certificate to A' in a natural way by adding a tag to `a`. In Section 4.2 we show the details of how we name these counterparts for each $\mathtt{a} : \mathtt{A}$ and each $\mathtt{b} : \mathtt{B}$. The type we produce, which corresponds to the disjunction $A \vee B$, is then not the union of the two input types but rather their *disjoint union*. We call this the 'coproduct type', $\mathtt{A} + \mathtt{B}$.

The need to define counterparts rather than allowing the same element to belong to multiple types may appear at first to be a disadvantage of type theory versus set theory. However, we show in Section 4.2 that an important feature of constructive logic follows from this requirement: when we have a certificate of $A \vee B$ we immediately know which of A or B it is a certificate to, since the procedure for constructing counterparts retains this information. Thus if we have a proof of a disjunction in constructive logic then we can immediately obtain from it a proof of one of the disjuncts. Conversely, to prove a disjunction we must prove one or other of the disjuncts. The same is not true in classical logic: we can classically

prove a disjunction without proving either disjunct.[32]

3.3 Negation

Classically the negation of a proposition P is true just in case that proposition is not true (and so is not true just in case proposition P is true). It follows that the conjunction of any proposition and its negation is never true, since one or other of its conjuncts must fail to be true. This is the Principle of Non-Contradiction and it is maintained in constructive logic.[33] This means that there should never be a certificate for a type that expresses a **contradiction**: consistency of the type theory requires that such types should have no tokens. This principle that a proposition and its negation cannot both be true is taken as the defining characteristic of negation in constructive logic.

We said in footnote 20 that there is one exception to the rule that any naming expression we can construct is the name of a token or a type: if we are given contradictory premises we can construct from them an *empty name*, i.e. an expression that is not the name of anything. For example, if we are given that some number n is a divisor of 9, and we are also given that the same n is an even number, then reasoning that begins from these contradictory premises may produce insensible mathematics and unmeaningful names.

This association goes both ways: contradictory premises are the only way we can construct empty names, and the way we demonstrate that our premises are contradictory is to derive an empty name from them. An empty name is one that would name a token of a type that has no tokens.

The simplest and most direct contradiction we can state is of course just the combination of a proposition and its negation. We can therefore take this as (the beginning of) a definition of negation: The negation of a proposition P is the proposition $\neg P$ that is, by definition, contradictory to P – i.e. the proposition that allows us to derive an empty name if we are also given P.

The simplest and most direct way to derive some conclusion from a given premise is to have a *function* that produces the conclusion as output when given the premise

[32] For example, let $R(a, b)$ denote that a and b are both irrational numbers and a^b is rational. Let $q = \sqrt{2}^{\sqrt{2}}$, so $q^{\sqrt{2}} = 2$. Then either q is rational and $R(\sqrt{2}, \sqrt{2})$, or q is irrational and $R(q, \sqrt{2})$. This argument (clasically) proves the disjunction $R(\sqrt{2}, \sqrt{2}) \vee R(q, \sqrt{2})$ without proving either disjunct.

[33] Other classical properties of negation such as the Law of Excluded Middle and the rule of Double Negation Elimination are not maintained in constructive logic. This is discussed further at the end of this section and in Section 5.

as input. We can therefore suppose that a certificate to $\neg P$ is a function that, when given a certificate to P, returns an empty name.

But which empty name? What should be the output type of such a function? As we noted in Section 1.3 there are many different empty types, i.e. descriptions that are not satisfied by any mathematical entity, such as 'even divisor of 9' or 'even prime number greater than 2'. We must choose one type to be the 'canonical' empty type for the purposes of defining negation – we don't want an infinite profusion of different negations of P, each purporting to produce a different empty name when given a certificate of P. Any empty type is as good as any other for the role of defining negation, and we have no criterion to prefer any one contradictory type over another.[34]

We therefore introduce a new type to play this role – a type that is by definition (intended to be) empty and has no other defining characteristics.[35] We call this the 'Zero Type', denoted $\mathtt{0}$.

The negation of a proposition P is then the type of functions from $\mathtt{P}$ to this Zero Type. That is, the proposition $\neg P$ corresponds to the type $\mathtt{P} \to \mathtt{0}$. From any function that purports to construct a particular empty name from a token of $\mathtt{P}$ we can construct a token of the negation type $\mathtt{P} \to \mathtt{0}$ (and vice versa). Thus, although the 'infinite profusion' of types mentioned above may still be defined, and all are inhabited exactly when $\mathtt{P} \to \mathtt{0}$ is, it is only the latter that is defined to be *the negation* of P. Thus we still have a single canonical negation of each proposition, not infinitely many distinct but logically equivalent ones.

This definition recovers exactly the properties of negation that we expect in a constructive logic. In particular, the absence of a token of $\mathtt{P}$ does not, in itself, guarantee that we can construct a function of type $\mathtt{P} \to \mathtt{0}$, and likewise the absence of such a function does not provide a way to construct a token of $\mathtt{P}$. That is, we may have *neither* a token of $\mathtt{P}$ nor a token of $\mathtt{P} \to \mathtt{0}$. This interpretation of negation therefore does not validate the Law of Excluded Middle (which says that for any proposition P the disjunction $P \vee \neg P$ is true), because as explained in the previous section we only have a certificate to a disjunction if we have a certificate to one or other of its disjuncts. Similarly the rule of Double Negation Elimination (which says that the negation of a negation of a proposition implies that proposition, i.e. $\neg\neg P \Rightarrow P$) is a not rule of constructive logic. This follows from the above

[34] Tradition has favoured statements such as '0 = 1' as a canonical example of a contradiction, but this is no more fundamental than any other candidate, and moreover requires the definition of the natural numbers and the equality sign, neither of which has been introduced yet!

[35] We cannot in fact assert that this type has no tokens, but its definition (Section 4.3) provides no direct way to form a token of this type. Consistency of the system then consists in the claim that no token of this type can be produced by any means.

interpretation, since from a function of type $(\mathtt{P} \to \mathtt{0}) \to \mathtt{0}$ we cannot in general derive a token of $\mathtt{P}$.

3.4 Truth and Falsity

Classically, for a given proposition P, there are exactly two possible situations: either P is true or $\neg P$ is true (in which case we say that P is *false*). In constructive logic, as we've seen, there are three possible (non-contradictory) situations: we have a certificate of P, we have a certificate of $\neg P$, or we have neither. How should we describe propositions in these various situations? Should we say that a proposition is 'false' whenever we are unable to produce a token of the corresponding type, or only when we have a token of the negation of the type? The first option is too strong: we want to leave open the interpretation that the proposition has a certificate that we are presently unable to obtain. On such an epistemic reading, lack of evidence for the truth of a proposition should not be conflated with evidence for the *lack of truth* of the proposition. We therefore reserve the word 'false' for propositions whose negations have certificates. That is, '$\mathtt{P}$ is false' will be a shorthand for '$\mathtt{P} \to \mathtt{0}$ is inhabited'.

What should we say about propositions for which we have neither a certificate to P nor a certificate to $\neg P$? Should we call them 'indeterminate', or say that that they 'don't have a truth value', or that their truth value is 'unknown'? Firstly, for the reasons discussed above, it would be too strong to say that such a proposition 'lacks a truth value' – we want to leave open the interpretation that one of P or $\neg P$ has a certificate that we are presently unable to obtain. More generally, we want to restrict our attention to things that can be expressed in the formal language – this is our way of ensuring that the things we say about mathematical entities are meaningful. But we have no way in this language to talk about the *absence* of a token of some type $\mathtt{P}$. The closest we could come is to say that the type's negation $\mathtt{P} \to \mathtt{0}$ is inhabited, but as we noted above this is stronger than the mere *absence* of a token of $\mathtt{P}$ and so does not capture the right notion. Since we have no way of expressing 'we have neither a token of $\mathtt{P}$ nor a token of $\mathtt{P} \to \mathtt{0}$' in the language, we do not introduce any special terminology for this, and we leave the reader to apply their own interpretation to the status of such propositions.

3.5 The Law of Explosion

In Section 3.3 we noted that reasoning beginning from contradictory premises "may produce insensible mathematics and unmeaningful names". This is a consequence of the Law of Explosion, a principle of classical logic that is maintained

in constructive logic, which says that from contradictory premises *any* conclusion follows. In the type theory we interpret this as: for any type `C` there is a function of type `0 → C`, which we call $\texttt{!}_{\texttt{C}}$.

Consequently, for any false proposition A (i.e. any proposition for which we have a token of `A → 0`) we can construct a function `A → C` by composing together the functions of type `A → 0` and `0 → C`. This explains the alternative name *ex falso quodlibet*, since from falsehood anything follows.

As we show in Section 5, Explosion is required in order to prove several useful classically valid principles – in particular the rule of 'Disjunctive Syllogism' (DS) that derives B from $A \vee B$ and $\neg A$, which we write as: $A \vee B,\ \neg A \vdash B$.

3.5.1 Justifying Explosion

Whereas the other features of the logic of HoTT have each been justified by thinking about the manipulation of certificates, it's not clear that the Law of Explosion can be similarly justified.

According to the definition given in Section 2.4, to define a function of type `0 → C` we require an expression of type `C` along with a variable `a` that is stipulated to be of type `0`. If (impossibly) we had a token of `0`, we could apply the function to it by replacing every instance of `a` in the expression (if there are any) with that token, resulting in an expression naming a token of `C`.

But the Law of Explosion does not specify an expression of type `C` to use, and does not require that we can provide such an expression. So the function $\texttt{!}_{\texttt{C}}$ doesn't appear to fit the template of Section 2, which is unsatisfactory.

We could choose to relax the definition in this case: since we know we will never in fact have a token of `0` to apply the function to, it arguably doesn't matter what expression of type `C` we use, or even whether we have such an expression at all.[36]

[36] This is essentially the justification given in the HoTT Book: for any type `C` "we can always construct a function `f : 0 → C` without having to give any defining equations, because there are no elements of `0` on which to define `f`." (p. 34)

A similar argument for adopting this rule proceeds as follows:

(i) there are no tokens of `0`, so it is impossible that we have such a token;

(ii) therefore, *if we had* a token of `0` we would be in an impossible contradictory situation;

(iii) from an impossible contradictory situation any consequence follows – in particular, the construction of a token of any type `C`;

(iv) thus, from a token of `0` we can produce a token of any type, so we always have a function of type $\mathtt{0} \to \mathtt{C}$

Of course, step (iii) in this argument is just the application of the principle of Explosion itself, but now at the meta-level at which we talk about what tokens we might have and what might be done with them (rather than within the framework of the theory itself). If we think that any theory worth considering must have a consistent meta-theory that endorses Explosion then the above argument might be acceptable.

But if we are trying to establish a new foundation for mathematics that incorporates its own logical foundation (rather than being built upon a separate logical framework) then, unless it is unavoidable, we should not assume the validity of rules in the meta-language in such a way that they are thereby imported into the system. We should instead be able to give justifications for the rules we assume and build in, in a way that follows from (or at least accords with) the basic intuitions that motivate the project.

One response to this would simply be to accept that Explosion is not justified, and therefore to discard it – to adopt a **paraconsistent** logic upon which to base a type theory.[37] However, this would also require us to abandon rules such as Disjunctive Syllogism mentioned above

$$A \vee B,\ \neg A \vdash B$$

since the only available justification of this rule proceeds via Explosion (see Section 5.6).

There may be some worth in pursuing an alternative approach grounded in paraconsistent logic, thereby building a type-theoretic foundation for logic that avoids Explosion and any rules from which it can be derived (and which arguably represents more faithfully the constructive intuitions upon which this project is founded).

[37] See `http://plato.stanford.edu/entries/logic-paraconsistent/`

However, such a project is beyond the scope of the present work which aims to explicate HoTT as it stands. Thus for the remainder of this Primer we assume that the Law of Explosion is indeed valid.[38]

3.6 Implication

Finally, we return to the motivating example for the Curry-Howard correspondence, namely the parallel between implication and functions and correspondingly between *modus ponens* and function application. The type corresponding to $A \Rightarrow B$ is the function type $\mathtt{A} \to \mathtt{B}$, and a certificate of $A \Rightarrow B$ is simply a function of this type. In this section we consider how this interpretation of $\Rightarrow$ behaves.

Since functions are defined as things that take an input of type `A` and construct from it an output of type `B`, we might expect the interpretation of $A \Rightarrow B$ in the type system to be something like that found in a *relevance logic*.[39] That is, we might imagine that $A \Rightarrow B$ will only hold if there is some particular connection of relevance between A and B that justifies why the truth of A implies the truth of B. However, as we will see, this is *not* the case: rather, $A \Rightarrow B$ is simply the material conditional.

Classically the material conditional is equivalent to $(\neg A) \vee B$ and to $\neg(A \,\&\, \neg B)$ and can therefore be defined in terms of the other logical operations. However, constructively we cannot assume that these equivalences hold.[40] Instead we take the material conditional to be defined by the following *truth table*:

A	B	$A \Rightarrow B$
F	F	T
F	T	T
T	F	F
T	T	T

where (per the discussion of Section 3.4) we interpret 'T' as 'the type is inhabited' and 'F' as 'the negation of the type is inhabited'.

We read the table from left to right, so for example we read the second line as: 'From a token of $\mathtt{A} \to \mathtt{0}$ and a token of `B` we can construct a token of $\mathtt{A} \to \mathtt{B}$', and

[38] In effect, we simply choose to regard $!_{\mathtt{C}} : \mathtt{0} \to \mathtt{C}$ as something that is *stipulated* to be a token of the function type, and that behaves just like a function defined in the usual way, even though it doesn't fit the usual definition.

[39] See `http://plato.stanford.edu/entries/logic-relevance/`

[40] In Section 5.9 we show that these are not constructively equivalent to $A \Rightarrow B$.

the third line as: 'From a token of $\mathtt{A}$ and a token of $\mathtt{B} \to \mathtt{0}$ we can construct a token of $(\mathtt{A} \to \mathtt{B}) \to \mathtt{0}$'.

Interpreted this way, it is easy to prove that all four lines of the truth table are satisfied:

(1) Given a function of type $\mathtt{A} \to \mathtt{0}$ we can compose this with the function $!_{\mathtt{B}} : \mathtt{0} \to \mathtt{B}$ given by the Law of Explosion to get a function of type $\mathtt{A} \to \mathtt{B}$.

(2) As we saw in Section 2.4, given a token of $\mathtt{B}$ we can construct a constant function of type $\mathtt{A} \to \mathtt{B}$.

(3) Given a token of $\mathtt{A}$ and a token of $\mathtt{B} \to \mathtt{0}$, if we had a function $\mathtt{A} \to \mathtt{B}$ we could combine it with these tokens to produce a contradiction, i.e. a token of $\mathtt{0}$. This can be formalised to define a function of type $(\mathtt{A} \to \mathtt{B}) \to \mathtt{0}$.

(4) As with (2), from a token of $\mathtt{B}$ we define a constant function of type $\mathtt{A} \to \mathtt{B}$.

We examine these arguments in more detail in Section 5.

3.7 The BHK interpretation

Let's summarise the chain of reasoning so far.

We start with the general notion that a proof in a constructive logic is a process that produces a certificate to the proposition being proved. This gives us the idea that certificates to propositions are mathematical objects, and this in turn gives us a type corresponding to each proposition P, which has the certificates to P as its tokens. We then take inspiration from the Curry-Howard correspondence that says that logical operations on propositions correspond to constructions in type theory. From these ideas we define a type theoretic construction corresponding to each of the logical operations by showing how to define a certificate to each compound proposition via the certificates to their component propositions.

From these considerations we obtain a version of the **BHK interpretation** of constructive (or intuitionistic) logic, defined by Brouwer, Heyting, and Kolmogorov:

- a certificate to $A \,\&\, B$ consists of a pair of certificates, the first being a certificate to A and the second being a certificate to B.
- a certificate to $A \vee B$ consists of either a certificate to A or a certificate to B, labelled to indicate which proposition it is a certificate for.
- a certificate to $A \Rightarrow B$ is a function that returns a certificate to B when given a certificate to A.
- a certificate to $\neg A$ is a function that returns a token of the Zero type when given a certificate to A.

To this we add the desideratum that we must be able to implement the resulting system with a computer program. This adds two further requirements:

- all functions are finitely specified procedures that are guaranteed to terminate within a finite number of computational steps.
- types are *intensional*, i.e. they are distinguished by their definitions, not by their contents.

This forms the conceptual basis for the type theory underlying HoTT.

In the following section we show exactly how these basic elements of the type theory are formally defined, and in Section 5 we give some examples of proofs using these tools and explore some ways in which the resulting constructive logic differs from classical logic. In subsequent sections we expand upon this basis to build up the remainder of the language of HoTT.

4 A Simple Type Theory

In the previous section we set out some of the basic requirements the type theory must satisfy in order for it to incorporate (constructive) logic under the Curry-Howard correspondence/BHK interpretation. Each of the logical connectives – implication, conjunction, disjunction, and negation – introduces a demand on the type theory: a new way of building types out of existing ones (or in the case of negation, a type `0` that corresponds to a 'canonical contradictory proposition'). We summarise these requirements as follows:

Logical operation	Logical notation	Type notation	Requirement
Implication	$A \Rightarrow B$	$\mathtt{A} \to \mathtt{B}$	Function Type
Conjunction	$A \,\&\, B$	$\mathtt{A} \times \mathtt{B}$	Product Type
Disjunction	$A \vee B$	$\mathtt{A} + \mathtt{B}$	Coproduct Type
Negation	$\neg A$	$\mathtt{A} \to \mathtt{0}$	Zero Type

In this section we explain more specific details of how we implement this by defining a very simple type system. This is not the full type theory of HoTT – in particular, the most basic types it provides do not correspond to interesting 'kinds of mathematical entities', but are almost trivial placeholders. But this very simple 'toy type theory' is useful to illustrate how we define the basic tools we need to do logical reasoning. We demonstrate this in Section 5 by going through a number of logical proofs in this system. Moreover, all the definitions introduced in this section will be retained in the full type system, so it is the central core around which more sophisticated tools are added.

We have seen in Section 2 how to define functions, which are the tokens of the function type $\mathtt{A} \to \mathtt{B}$. We use these in stating the definitions of other types, and thereby avoid having to directly manipulate expressions.

To define a new type – a new way of taking old types and forming new ones from them – we need to provide the following:

- a **type former** that gives us a way of *naming* the new type (in a way that may depend upon the names of the input types, if there are any);
- zero, one, or more **token constructor**s – functions that produce a token of the new type (given some tokens of the input types, if there are any);
- **elimination rule**s that tell us how to use tokens of the new type, i.e. how to define functions that take tokens of this type as input.

We sometimes also provide **computation rules** that tell us how the elimination rules and constructors interact.

4.1 Product Types

We said in Section 3.1 that to implement logical conjunction $A \,\&\, B$ we need a type whose tokens are pairs of certificates to A and B. We therefore introduce the **product type** to play this role. The product of types `A` and `B` is written $\mathtt{A} \times \mathtt{B}$. That is, if expressions '`A`' and '`B`' name types, then the expression '$\mathtt{A} \times \mathtt{B}$' also names a type as well.

Since types are defined intensionally (i.e. they are distinguished by their descriptions) we must consider $\mathtt{A} \times \mathtt{B}$ and $\mathtt{B} \times \mathtt{A}$ to be two *distinct* types.

4.1.1 Token constructor for products

To construct a token of a product type we need a token of each of the two input types; we write the token of the product type as $(\mathtt{a}, \mathtt{b})$. This is analogous to the logical rule of $\&$**-introduction**: $A, B \vdash A \,\&\, B$.

Since $\mathtt{A} \times \mathtt{B}$ and $\mathtt{B} \times \mathtt{A}$ are distinct types, we should think of the tokens of $\mathtt{A} \times \mathtt{B}$ as *ordered* pairs: if $(\mathtt{a}, \mathtt{b})$ is a token of $\mathtt{A} \times \mathtt{B}$ then $(\mathtt{b}, \mathtt{a})$ is a token of $\mathtt{B} \times \mathtt{A}$. It is clear that from any token of $\mathtt{A} \times \mathtt{B}$ we can construct a token of $\mathtt{B} \times \mathtt{A}$.

Note also that these ordered pairs are not built out of other things, in the way that ordered pairs in set theory are built up (for example as Kuratowski ordered pairs $\{\{a\}, \{a, b\}\}$) – they are *primitive*.

4.1.2 Elimination rule for products

If we are given a token of a product type, how do we use it? Since types correspond to propositions and tokens are certificates to those propositions, to *use* a token of a product type is to *carry out a deduction* from the conjunction. At the minimum, then, we need to know how to apply *modus ponens* where we have a conjunction as the antecedent. This means we need to know how to define a function from $\mathtt{A} \times \mathtt{B}$ to some type `C`.

We've seen in Section 2.6 how to define functions of multiple inputs via *currying* – a function that takes two inputs `a` and `b` is written as a function that takes `a` as its input and returns a new function, which in turn takes `b` as input and returns an output of the appropriate type.

In the present case, to define a function of type $(\mathtt{A} \times \mathtt{B}) \to \mathtt{C}$ we need a curried function $\mathtt{f} : \mathtt{A} \to (\mathtt{B} \to \mathtt{C})$. Given this we can define $\mathtt{g} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{C}$, which works by first applying `f` to `a` to produce a function $\mathtt{f_a} : (\mathtt{B} \to \mathtt{C})$, and then applying this $\mathtt{f_a}$ to `b` to give $\mathtt{f_a}(\mathtt{b}) : \mathtt{C}$.

This process, the reverse of currying, is called **uncurrying**. We show in Section 5.4 that there is a one-to-one correspondence between curried and uncurried functions – each curried function gives us an uncurried version as described above, and all functions of type $(\mathtt{A} \times \mathtt{B}) \to \mathtt{C}$ are obtained in this way.

Currying and uncurrying correspond to the two directions of the logical equivalence

$$(A \,\&\, B) \Rightarrow C \quad \text{iff} \quad A \Rightarrow (B \Rightarrow C)$$

4.1.3 Projectors

As a first example we consider the two **projectors** that respectively return the first or second component of a given pair:

$$\mathtt{pr}_{\mathtt{A}} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{A} \quad \text{and} \quad \mathtt{pr}_{\mathtt{B}} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{B}$$

We define these by giving curried functions

$$\mathtt{pr}'_{\mathtt{A}} : \mathtt{A} \to (\mathtt{B} \to \mathtt{A}) \quad \text{and} \quad \mathtt{pr}'_{\mathtt{B}} : \mathtt{A} \to (\mathtt{B} \to \mathtt{B})$$

In each case there is only one function of the appropriate type that is well-defined in general for all types A and B. These are the trivial *constant* and *identity* functions that we considered in Section 2.4.

- Given any token $\mathtt{a} : \mathtt{A}$ we can define a constant function $[\mathtt{b} \mapsto \mathtt{a}] : \mathtt{B} \to \mathtt{A}$ that ignores its input and always returns a. The function $\mathtt{pr}'_{\mathtt{A}}$ must therefore take a as input and return this constant function, so $\mathtt{pr}'_{\mathtt{A}} :\equiv [\mathtt{a} \mapsto [\mathtt{b} \mapsto \mathtt{a}]]$.
- For any type B we can define the identity function $\mathtt{id}_{\mathtt{B}}$ that returns its inputs unchanged. $\mathtt{pr}'_{\mathtt{B}}$ must be the function that ignores its input and always returns $\mathtt{id}_{\mathtt{B}}$. So $\mathtt{pr}'_{\mathtt{B}} :\equiv [\mathtt{a} \mapsto [\mathtt{b} \mapsto \mathtt{b}]]$.

It is then a simple exercise to see that the functions we get by uncurrying $\mathtt{pr}'_{\mathtt{A}}$ and $\mathtt{pr}'_{\mathtt{B}}$ are exactly the projectors that pick out the first and second components of a pair. These projectors are the certificates to the propositions $A \,\&\, B \Rightarrow A$ and $A \,\&\, B \Rightarrow B$ that correspond to the $\&$**-elimination** rules.

4.2 Coproduct Types

We said in Section 3.2 that to implement logical disjunction we need, for any two types A and B, a type $\mathtt{A} + \mathtt{B}$ called the 'coproduct' of A and B. That is, if expressions '$\mathtt{A}$' and '$\mathtt{B}$' name types then the expression '$\mathtt{A} + \mathtt{B}$' also names a type as well.

A token of `A + B` is (a counterpart to) *either* a token of `A` *or* a token of `B`. If we are given a token of the coproduct we will always be able to see which of the two input types it came from. Thus the coproduct of two types is analogous to the disjoint union of two sets.

4.2.1 Token constructor for coproducts

Unlike the product type, the coproduct type has *two* token constructors, since we must be able to produce a token of `A + B` if we are given a token of `A` or a token of `B`. We call the constructor that 'injects' tokens of `A` into `A + B` the 'left injection'

$$\texttt{inl} : \texttt{A} \to \texttt{A} + \texttt{B}$$

and similarly we have the 'right injection' that injects tokens of `B`

$$\texttt{inr} : \texttt{B} \to \texttt{A} + \texttt{B}$$

That is, if '`a`' names a token of `A` then '`inl(a)`' names a token of `A + B`, and likewise if '`b`' names a token of `A` then '`inr(b)`' names a token of `A + B`. We have no further way of reducing these expressions – there is no further substitution we can do to eliminate the labels `inl` and `inr`.

This is the 'natural way' that we distinguish tokens of the component types from their counterpart tokens in the coproduct, as discussed in Section 3.2. We simply retain the names of the injection functions as part of the names of the counterpart tokens, thereby producing a distinct (but obviously closely related) token that records the information about which of the two component types the token came from.[41]

As an alternative notation we sometimes write '$\langle$`a`]' for `inl(a)` and '[`b`$\rangle$' for `inr(b)`.

4.2.2 Elimination rule for coproducts

To define a function from `A + B` to some `C` we must have functions that can take inputs of the form `inl(a)` and `inr(b)`, i.e. functions $\texttt{g}_l : \texttt{A} \to \texttt{C}$ and $\texttt{g}_r : \texttt{B} \to \texttt{C}$. A function $\texttt{f} : (\texttt{A} + \texttt{B}) \to \texttt{C}$ must then examine its input, see whether it came from `A` or from `B`, and then apply one or other of $\texttt{g}_l$ or $\texttt{g}_r$ accordingly. This is called **case analysis**, and is used whenever we have multiple token constructors for a type.

[41] Strictly, the injectors ought to be written as '$\texttt{inl}^{\texttt{A+B}}$' and '$\texttt{inr}^{\texttt{A+B}}$' to distinguish the constructors of one coproduct from those of another: $\texttt{inl}^{\texttt{A+B}}\texttt{(a)}$ is a token of `A + B`, and therefore distinct from $\texttt{inl}^{\texttt{A+Z}}\texttt{(a)}$, which is a token of `A + Z`, but omitting these superscripts obscures that. However, in practice we will abuse notation for the sake of simplicity, and omit the superscripts whenever the coproduct involved is unambiguous from context.

Thus to define a function $\mathtt{f} : (\mathtt{A} + \mathtt{B}) \to \mathtt{C}$ we just need to specify functions $\mathtt{g}_l$ and $\mathtt{g}_r$ of the appropriate types. The function $\mathtt{f}$ is then defined by case analysis as:

$$\mathtt{f}(\mathtt{inl}(\mathtt{a})) :\equiv \mathtt{g}_l(\mathtt{a})$$

$$\mathtt{f}(\mathtt{inr}(\mathtt{b})) :\equiv \mathtt{g}_r(\mathtt{b})$$

4.3 The Zero Type

We can give the definition of the Zero Type very briefly, since it corresponds to a contradictory proposition which is intended to have no certificates. The main use of the Zero type is in the definition of negation.

Since the Zero type doesn't depend upon any other types we don't have a type former – we just have the name of the type itself, $\mathtt{0}$ (which we could think of as a type former that takes no inputs).

By definition $\mathtt{0}$ has no token constructors.

The elimination rule for the Zero type is the Law of Explosion discussed in Section 3.5, which says that for any type $\mathtt{C}$ we always have a function of type $\mathtt{0} \to \mathtt{C}$. We call this function $!_{\mathtt{C}} : \mathtt{0} \to \mathtt{C}$.

4.4 The Unit Type

The above definitions are sufficient for propositional logic, as we show in Section 5. We have specified ways to construct new types from old ones, and to construct tokens of the new types from tokens of the old ones. But none of them entail that there are any inhabited types at all.[42] Hence, they are compatible with what we might call a 'trivial interpretation' in which there are (almost) no inhabited types. To avoid this trivial interpretation we introduce one additional type that is inhabited *by definition*.

We define the **Unit type**, written as $\mathtt{1}$, as a type having just a single token constructor, which produces a token that we write as $* : \mathtt{1}$.

To define a function $\mathtt{f} : \mathtt{1} \to \mathtt{C}$ for any $\mathtt{C}$ we simply require a single token $\mathtt{c} : \mathtt{C}$ to serve as the output for this function, so that we can define $\mathtt{f}(*) :\equiv \mathtt{c}$. (Such a function is therefore a constant function as discussed in Section 2.4.) Note that for any type $\mathtt{A}$ we can define the constant function $\mathtt{k}_* : \mathtt{A} \to \mathtt{1}$ that sends any token $\mathtt{a} : \mathtt{A}$ to $* : \mathtt{1}$.

[42] This is not quite true: there must at least be an identity function $\mathrm{id}_{\mathtt{0}} : \mathtt{0} \to \mathtt{0}$, and tokens of other related types constructed from $\mathtt{0} \to \mathtt{0}$.

5 Doing Logic in Type Theory

In Section 4 we set up enough machinery to do some simple logical deductions in the type theory. In this section we investigate some classical theorems to see how to prove them in HoTT. Some proofs that are classically valid cannot be replicated in constructive logic nor can alternative proofs be given. We therefore need to take care when reasoning, in order to ensure that we are not relying on habitual classical inferences that are not constructively valid. In this section we build up a library of constructively valid moves that can be used to simplify subsequent proofs.

Part of the goal of this section is to illustrate and reinforce the interpretation of tokens and types used throughout the previous sections, namely that types correspond to propositions and tokens correspond to certificates to propositions. This way of thinking about logic is useful because constructive logic follows very naturally from it, as we saw in Section 3. It is therefore easier to re-train intuitions about logical reasoning from classical to constructive thinking by using this model. It is also helpful when introducing new elements into the type theory, such as *quantifiers* and *identity types*, since it's easier to understand them and motivate their definitions with this explicitly semantic tokens-as-certificates understanding.

Of course we could define purely *syntactic* rules (a system like Natural Deduction, for example) and express all our proofs as formal manipulations of expressions.[43] This would serve to show which proofs are constructively valid and which are not, but it would provide no experience of thinking about proofs as manipulations of certificates to propositions. In the following proofs we will therefore work in a semantic mode, using the language of 'being given certificates' and 'constructing certificates'.

5.1 Notation

First we define some notation. To illustrate the translation we write each example in two forms: first using the familiar 'logical notation' ($\&$, $\vee$, $\Rightarrow$, and $\neg$) and then in the language of the type theory defined in the previous section ($\times$, $+$, $\to$, and $\to 0$).

The theorems to be proved will be of the form 'if Q and R and ..., then P', which we write as:

$$Q,\ R, \ldots \vdash P$$

where P is a single proposition (the 'conclusion') and Q, R, etc. may be any

[43] For a formal deduction system using introduction and elimination rules, see Appendix A.2 of the HoTT Book.

(finite) number of propositions (the 'premises'). In particular, if the theorem is a tautology involving no premises we write $\vdash P$.

The translation of these into type theory is usually written in the same way:

$$\mathtt{Q},\ \mathtt{R}, \ldots \vdash \mathtt{P}$$

(and $\vdash \mathtt{P}$ for tautologies). However, in the constructive type theory this should be read as 'given tokens of types $\mathtt{Q}$, $\mathtt{R}$, ... we can construct a token of type $\mathtt{P}$'. It is often useful to have names for these tokens that we can use in the body of the proof, so we sometimes write the theorems as

$$\mathtt{q} : \mathtt{Q},\ \mathtt{r} : \mathtt{R}, \ldots \vdash \mathtt{p} : \mathtt{P}$$

and sometimes we only name the tokens to which we need to refer. When we have tokens of negation types like $\mathtt{A} \to \mathtt{0}$ we often use the convention of naming them with an overbar, e.g. $\bar{\mathtt{a}} : \mathtt{A} \to \mathtt{0}$. Similarly, tokens of double negation types like $(\mathtt{A} \to \mathtt{0}) \to \mathtt{0}$ are sometimes given names with double overbars, e.g. $\bar{\bar{\mathtt{a}}}$.

If we have $\mathtt{A} \vdash \mathtt{B}$ and $\mathtt{B} \vdash \mathtt{A}$ then we write $\mathtt{A} \dashv\vdash \mathtt{B}$. Sometimes (usually for comparison with another theorem) we write $\mathtt{B} \vdash \mathtt{A}$ as $\mathtt{A} \dashv \mathtt{B}$.

We might also want to express facts about *relative constructability* – if we can construct $\mathtt{B}$ from $\mathtt{A}$ then we can also construct $\mathtt{Y}$ from $\mathtt{X}$. We write this as

$$\frac{\mathtt{A} \vdash \mathtt{B}}{\mathtt{X} \vdash \mathtt{Y}}$$

So, for example, we have

$$\frac{\mathtt{A} \vdash \mathtt{B} \quad \mathtt{B} \vdash \mathtt{C}}{\mathtt{A} \vdash \mathtt{C}}$$

As we show in the next section, this relative constructability notation does not provide more expressive power, since it turns out to be just a convenient notation for something we could already express without it (i.e. what in computer science would be called 'syntactic sugar'.)

5.2 How do we construct a token?

A proof in the type theory involves the construction a token of some specified type. We should therefore review how we construct tokens of the various types we've encountered.

Sometimes we can recover the token we want from another token:

- We may have a function available – either given in the premises or one that we've constructed – that returns a token of the type we want when given an appropriate input. In that case we have reduced the problem to producing a token of that function's input type (which may or may not be easier than the original problem).

- We may have a token of a product type available that contains as one of its components a token of the type we're looking for. In that case, we just extract it with the appropriate projector (which is always available).

- We may have a token of a coproduct that has as one of its components a token of the type we want. In that case, we still have work to do. By case analysis on the coproduct we can say that *if* the token we have is from the appropriate side of the coproduct then it's the one we want. But then we have to go on to *prove* that it is from that side and not the other. If we can't do that, then we've reduced the problem to constructing the token we want from a token of the *other* side of this coproduct (which may or may not be easier than the original problem).

If we can't do any of these, we have to construct the token we want:

- To construct a token of a product type, such as $\mathtt{A} \times \mathtt{B}$, we must construct tokens of each of the component types, $\mathtt{a} : \mathtt{A}$ and $\mathtt{b} : \mathtt{B}$.

- To construct a token of a coproduct type, such as $\mathtt{A} + \mathtt{B}$, we must construct a token of one of the component types, either $\mathtt{a} : \mathtt{A}$ or $\mathtt{b} : \mathtt{B}$ – but either will suffice.

If we need to construct a token of a function type (i.e. a function) then, aside from the methods mentioned above, there are basically two ways to do this.

- We may have two functions already available that can be composed together to produce the function we want. That is, if we're trying to construct a function of some type $\mathtt{A} \to \mathtt{C}$, then we may have some functions $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ and $\mathtt{g} : \mathtt{B} \to \mathtt{C}$ (for some particular type $\mathtt{B}$) in which case $\mathtt{g} \circ \mathtt{f}$ gives us the function we want.

- If we can't do that, our remaining option is to use the HoTT analogue of the **Deduction theorem** which says that if we can produce a token of type $\mathtt{Z}$ given some premises along with a token of $\mathtt{Y}$, then we can produce a token of the type $\mathtt{Y} \to \mathtt{Z}$, given those premises.

In natural deduction the **Deduction theorem** says that if

$$\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots, \mathtt{Y} \vdash \mathtt{Z}$$

then

$$\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots \vdash \mathtt{Y} \to \mathtt{Z}$$

In the notation defined above, the Deduction theorem is written:

$$\frac{\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots, \mathtt{Y} \vdash \mathtt{Z}}{\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots \vdash \mathtt{Y} \to \mathtt{Z}}$$

This holds because to construct a function of type $\mathtt{Y} \to \mathtt{Z}$ from inputs of types $\mathtt{A}$, $\mathtt{B}$, $\mathtt{C}\ldots$, we need to say what it does when it's given any token of type $\mathtt{Y}$. That is, given an *arbitrary* token of $\mathtt{Y}$ (together with the other inputs) we must specify how to produce a token of $\mathtt{Z}$ which is exactly what the first line says we can do.

It is important that the token of type $\mathtt{Y}$ we assume to be given is *arbitrary* – we can't assume anything special about it (e.g. if it's a coproduct type, we can't assume that it's from one side of the coproduct rather than the other).

The converse of the Deduction Theorem is called **Cut**.

$$\frac{\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots \vdash \mathtt{Y} \to \mathtt{Z}}{\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots, \mathtt{Y} \vdash \mathtt{Z}}$$

This obviously holds in HoTT, because if we have the function type and a token of its input type we can apply the function to get a token of the output type.

Note that when we are trying to derive a negated proposition $\neg P$ from some premises, this corresponds to constructing a function of type $\mathtt{P} \to \mathtt{0}$. According to the Deduction theorem, to do this it is sufficient to add $\mathtt{P}$ to the premises and derive a contradiction, since we have

$$\frac{\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots, \mathtt{P} \vdash \mathtt{0}}{\mathtt{A},\ \mathtt{B},\ \mathtt{C}, \ldots \vdash \mathtt{P} \to \mathtt{0}}$$

This is exactly the constructively valid method of *Reductio ad absurdum*.

Note that although *Reductio ad absurdum* is constructively valid, the *Principle of Indirect Proof* is not: adding $\neg P$ to the premises and deriving a contradiction doesn't give a constructive proof of P; rather, it gives a proof of $\neg\neg P$, which is constructively weaker.

By the Deduction theorem, whenever we have $\mathtt{A} \vdash \mathtt{B}$ we also have $\vdash \mathtt{A} \to \mathtt{B}$, and so whenever we prove relative constructability

$$\frac{\mathtt{A} \vdash \mathtt{B}}{\mathtt{X} \vdash \mathtt{Y}}$$

this is equivalent to proving

$$\mathtt{A} \to \mathtt{B},\ \mathtt{X} \vdash \mathtt{Y}$$

Thus the relative constructability notation is just a more convenient way of expressing something we could already say without it (except in the statement of the Deduction theorem itself). But since it is more convenient, and aligns nicely with a useful way of thinking about constructions, we use it in subsequent proofs.

5.3 Rules involving & and ∨

Some of the basic rules of classical logic are taken as part of the definition of the logical connectives in HoTT, and so we know that they hold. Some others follow very simply from the definitions we've given.

The Introduction and Elimination rules for conjunction and disjunction were used in the definitions of the product and coproduct, and so they hold automatically:

&-introduction

Theorem 1 (&-introduction)**.**

$$A,\ B \vdash A \,\&\, B$$
$$\mathtt{a} : \mathtt{A},\ \mathtt{b} : \mathtt{B} \vdash \mathtt{A} \times \mathtt{B}$$

Proof. The definition of the product type (Section 4.1) says that a token of $\mathtt{A} \times \mathtt{B}$ consists of a pair $(\mathtt{a}, \mathtt{b})$, where $\mathtt{a} : \mathtt{A}$ and $\mathtt{b} : \mathtt{B}$, so &-introduction corresponds to the token constructor for product types.

Theorem 2 (&-elimination)**.**

$$A \,\&\, B \vdash A \qquad\qquad A \,\&\, B \vdash B$$
$$(\mathtt{a},\ \mathtt{b}) : \mathtt{A} \times \mathtt{B} \vdash \mathtt{A} \qquad\qquad (\mathtt{a},\ \mathtt{b}) : \mathtt{A} \times \mathtt{B} \vdash \mathtt{B}$$

Proof. These correspond to the *projectors* $\mathtt{pr}_1 : \mathtt{A}\times\mathtt{B} \to \mathtt{A}$ and $\mathtt{pr}_2 : \mathtt{A}\times\mathtt{B} \to \mathtt{B}$ (defined in Section 4.1.3) which extract the first and second component respectively from a token $(\mathtt{a}, \mathtt{b}) : \mathtt{A} \times \mathtt{B}$.

Theorem 3 (∨-introduction)**.**

$$A \vdash A \vee B \qquad\qquad B \vdash A \vee B$$
$$\mathtt{a} : \mathtt{A} \vdash \mathtt{inl}(\mathtt{a}) : \mathtt{A} + \mathtt{B} \qquad\qquad \mathtt{b} : \mathtt{B} \vdash \mathtt{inr}(\mathtt{b}) : \mathtt{A} + \mathtt{B}$$

Proof. These are the token constructors in the definition of the coproduct type.

Theorem 4 (∨-elimination)**.**

$$A \Rightarrow Z,\ B \Rightarrow Z \ \vdash\ A \vee B \Rightarrow Z$$
$$\mathtt{g}_l : \mathtt{A} \to \mathtt{Z},\ \mathtt{g}_r : \mathtt{B} \to \mathtt{Z} \ \vdash\ \mathtt{f} : (\mathtt{A} + \mathtt{B}) \to \mathtt{Z}$$

Proof. This is the elimination rule in the definition of the coproduct type.

Note that if we apply Cut to this we recover the more usual form:

$$A \Rightarrow Z,\ B \Rightarrow Z,\ A \vee B \vdash\ Z$$

Some other rules are so trivial that they can be checked immediately by the reader, and so we won't write them out here. In particular, the reader is left to verify that conjunction is:

Commutative $A \,\&\, B \dashv\vdash B \,\&\, A$

Associative $(A \,\&\, B) \,\&\, C \dashv\vdash A \,\&\, (B \,\&\, C)$

Idempotent $A \,\&\, A \dashv\vdash A$

and likewise for disjunction.

It is also easy to show that the basic classical rules for the interaction between & and $\vee$, namely Absorption and Distributivity hold constructively.

Theorem 5 (Absorption1)**.**

$$A \vee (A \,\&\, B) \dashv\vdash A$$
$$\mathtt{A} + (\mathtt{A} \times \mathtt{B}) \dashv\vdash \mathtt{A}$$

Proof. The right-to-left direction is trivial: given $\mathtt{a} : \mathtt{A}$ we have $\langle \mathtt{a}] : \mathtt{A} + (\mathtt{A} \times \mathtt{B})$. For the left-to-right direction, we do case analysis on the coproduct: if we have $\langle \mathtt{a}]$ then we have a token of $\mathtt{A}$; if we have $[(\mathtt{a}, \mathtt{b})\rangle$ then the projector $\mathtt{pr}_1$ gives the token of $\mathtt{A}$.

Theorem 6 (Absorption2)**.**

$$A \,\&\, (A \vee B) \dashv\vdash A$$
$$\mathtt{A} \times (\mathtt{A} + \mathtt{B}) \dashv\vdash \mathtt{A}$$

Proof. The left-to-right direction is given by the projector from the product. For the right-to-left direction, given $\mathtt{a} : \mathtt{A}$ we can form $(\mathtt{a}, \langle \mathtt{a}]) : \mathtt{A} \times (\mathtt{A} + \mathtt{B})$.

Theorem 7 (Distributivity of $\vee$ over &)**.**

$$A \vee (B \,\&\, C) \dashv\vdash (A \vee B) \,\&\, (A \vee C)$$
$$\mathtt{A} + (\mathtt{B} \times \mathtt{C}) \dashv\vdash (\mathtt{A} + \mathtt{B}) \times (\mathtt{A} + \mathtt{C})$$

Proof. For the left-to-right direction we do case analysis on the coproduct:

- if we have a token of $\mathtt{A}$ then we have $(\langle \mathtt{a}], \langle \mathtt{a}]) : (\mathtt{A} + \mathtt{B}) \times (\mathtt{A} + \mathtt{C})$.
- if we have a token of $(\mathtt{B} \times \mathtt{C})$ then the projectors give us $\mathtt{b} : \mathtt{B}$ and $\mathtt{c} : \mathtt{C}$, and so we can produce $([\mathtt{b}\rangle, [\mathtt{c}\rangle) : (\mathtt{A} + \mathtt{B}) \times (\mathtt{A} + \mathtt{C})$.

For the right-to-left direction, we can do two simultaneous case analyses on the two coproducts, giving four possibilities.[44] The token of the product we're given is then either $\big(\langle \mathtt{a}], \langle \mathtt{a}]\big)$, $\big(\langle \mathtt{a}], [\mathtt{c}\rangle\big)$, $\big([\mathtt{b}\rangle, \langle \mathtt{a}]\big)$, or $\big([\mathtt{b}\rangle, [\mathtt{c}\rangle\big)$ (for some $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$ of the respective types). In the first three cases we have a token of $\mathtt{A}$, and so we can produce $\langle \mathtt{a}] : \mathtt{A} + (\mathtt{B} \times \mathtt{C})$. In the fourth case we have a token of $\mathtt{B}$ and a token of $\mathtt{C}$, so we can can produce $[(\mathtt{b}, \mathtt{c})\rangle : \mathtt{A} + (\mathtt{B} \times \mathtt{C})$. So in any case we can produce a token of the required coproduct.

Theorem 8 (Distributivity of & over ∨)**.**

$$A \,\&\, (B \vee C) \dashv\vdash (A \,\&\, B) \vee (A \,\&\, C)$$
$$\mathtt{A} \times (\mathtt{B} + \mathtt{C}) \dashv\vdash (\mathtt{A} \times \mathtt{B}) + (\mathtt{A} \times \mathtt{C})$$

Proof. In each direction we can do case analysis on the coproduct we're given. We always have either tokens $\mathtt{a}$ and $\mathtt{b}$ or tokens $\mathtt{a}$ and $\mathtt{c}$, from which we can construct the token required.

5.4 Rules involving ⇒

The correspondence between *modus ponens* and function application is part of the motivation for using the Curry-Howard correspondence to incorporate logic into the type theory, and so naturally it holds by definition:

Theorem 9 (Modus ponens)**.**

$$A,\ A \Rightarrow B \vdash B$$
$$\mathtt{a} : \mathtt{A},\ \mathtt{f} : \mathtt{A} \to \mathtt{B} \vdash \mathtt{f}(\mathtt{a}) : \mathtt{B}$$

Proof. Given a function $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ and a token $\mathtt{a} : \mathtt{A}$, the result of applying $\mathtt{f}$ to $\mathtt{a}$ is, by definition, a token $\mathtt{f}(\mathtt{a}) : \mathtt{B}$.

Theorem 10 (Transitivity of ⇒)**.**

$$A \Rightarrow B,\ B \Rightarrow C \vdash A \Rightarrow C$$
$$\mathtt{f} : \mathtt{A} \to \mathtt{B},\ \mathtt{g} : \mathtt{B} \to \mathtt{C} \vdash \mathtt{g} \circ \mathtt{f} : \mathtt{A} \to \mathtt{C}$$

Proof. Transitivity of ⇒ corresponds to composition of functions.

In Section 4.1.2 we used Currying and Uncurrying to define functions from product types. Uncurrying says that, given a function of type $\mathtt{A} \to (\mathtt{B} \to \mathtt{C})$, we can define a

[44] Alternatively we could write these as nested case analyses, first analysing the first coproduct and then, for each case of that, doing a case analysis of the second.

function of type $(\mathtt{A} \times \mathtt{B}) \to \mathtt{C}$ which takes tokens of the product type as its inputs. Currying says that any function from the product type has a corresponding curried version.

Theorem 11 (Currying)**.**

$$(A \,\&\, B) \Rightarrow C \vdash A \Rightarrow (B \Rightarrow C)$$
$$\mathtt{f} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{C} \vdash \mathtt{g} : \mathtt{A} \to (\mathtt{B} \to \mathtt{C})$$

Proof. Given $\mathtt{f}$ as above, for any $\mathtt{a} : \mathtt{A}$ the function $\mathtt{g}_\mathtt{a}$ should map any $\mathtt{b} : \mathtt{B}$ to $\mathtt{f}((\mathtt{a}, \mathtt{b}))$. We can therefore define $\mathtt{g}$ as $\mathtt{g}(\mathtt{a})(\mathtt{b}) :\equiv \mathtt{f}((\mathtt{a}, \mathtt{b}))$.

Theorem 12 (Uncurrying)**.**

$$A \Rightarrow (B \Rightarrow C) \vdash (A \,\&\, B) \Rightarrow C$$
$$\mathtt{g} : \mathtt{A} \to (\mathtt{B} \to \mathtt{C}) \vdash \mathtt{f} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{C}$$

Proof. Given a function $\mathtt{g}$ then we can define $\mathtt{f}$ as follows: for any token of $\mathtt{A} \times \mathtt{B}$, the projectors $\mathtt{pr}_1$ and $\mathtt{pr}_2$ give tokens $\mathtt{a} : \mathtt{A}$ and $\mathtt{b} : \mathtt{B}$. Applying $\mathtt{g}$ to $\mathtt{a}$ gives a function $\mathtt{g}_\mathtt{a} : \mathtt{B} \to \mathtt{C}$, and applying this to $\mathtt{b}$ gives a token of $\mathtt{C}$. We can therefore define $\mathtt{f}$ as $\mathtt{f}((\mathtt{a}, \mathtt{b})) :\equiv \mathtt{g}(\mathtt{a})(\mathtt{b})$.

It's clear by inspection that currying and uncurrying are inverses of each other.

Theorem 13 (Swapping the order of function arguments)**.**

$$A \Rightarrow (B \Rightarrow C) \vdash B \Rightarrow (A \Rightarrow C)$$
$$\mathtt{A} \to (\mathtt{B} \to \mathtt{C}) \vdash \mathtt{B} \to (\mathtt{A} \to \mathtt{C})$$

Proof. This can be proved by an application of Uncurrying, then Commutativity of Conjunction, and then Currying.

Finally, we give a more complicated example as an application of the Deduction theorem (Section 5.2):

Theorem 14.

$$A \Rightarrow (B \Rightarrow C) \;\vdash\; (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$$
$$\mathtt{g} : \mathtt{A} \to (\mathtt{B} \to \mathtt{C}) \;\vdash\; \mathtt{h} : (\mathtt{A} \to \mathtt{B}) \to (\mathtt{A} \to \mathtt{C})$$

Proof. By the Deduction Theorem we have

$$\frac{\mathtt{A} \to (\mathtt{B} \to \mathtt{C}),\ \mathtt{A} \to \mathtt{B} \;\vdash\; \mathtt{A} \to \mathtt{C}}{\mathtt{A} \to (\mathtt{B} \to \mathtt{C}) \;\vdash\; (\mathtt{A} \to \mathtt{B}) \to (\mathtt{A} \to \mathtt{C})}$$

and applying it again gives

$$\frac{\mathtt{A} \to (\mathtt{B} \to \mathtt{C}),\ \mathtt{A} \to \mathtt{B},\ \mathtt{A} \vdash \mathtt{C}}{\mathtt{A} \to (\mathtt{B} \to \mathtt{C}),\ \mathtt{A} \to \mathtt{B} \vdash \mathtt{A} \to \mathtt{C}}$$

This top line is easily proved: given a token $\mathtt{a} : \mathtt{A}$ and a function $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ we have $\mathtt{f}(\mathtt{a}) : \mathtt{B}$. Given a function $\mathtt{g} : \mathtt{A} \to (\mathtt{B} \to \mathtt{C})$ we can produce $\mathtt{g}(\mathtt{a}) : \mathtt{B} \to \mathtt{C}$. Putting these together gives $\mathtt{g}(\mathtt{a})(\mathtt{f}(\mathtt{a})) : \mathtt{C}$, as required.

Thus the function $\mathtt{h} : (\mathtt{A} \to \mathtt{B}) \to (\mathtt{A} \to \mathtt{C})$ that we were originally required to define, given a function $\mathtt{g} : \mathtt{A} \to (\mathtt{B} \to \mathtt{C})$, can be written as $\mathtt{h} :\equiv \big[\mathtt{f} \mapsto [\mathtt{a} \mapsto \mathtt{g}(\mathtt{a})(\mathtt{f}(\mathtt{a}))]\big]$.

5.5 Rules involving Negation

Since the type corresponding to $\neg P$ is the function type $\mathtt{P} \to \mathtt{0}$, many of the rules involving negation are just special cases of the rules proved above.

The Principle of Non-Contradiction is definitive of negation and is easily verified:

Theorem 15 (Non-Contradiction)**.**

$$\vdash \neg(P \,\&\, \neg P)$$
$$\vdash \mathtt{P} \times (\mathtt{P} \to \mathtt{0}) \to \mathtt{0}$$

Proof. A token of $\mathtt{P} \times (\mathtt{P} \to \mathtt{0})$ is a pair $(\mathtt{p} : \mathtt{P},\ \bar{\mathtt{p}} : \mathtt{P} \to \mathtt{0})$. So by function application, we have $\bar{\mathtt{p}}(\mathtt{p}) : \mathtt{0}$. That is, non-contradiction is an instance of *modus ponens*.

Theorem 16 (Modus Tollens)**.**

$$A \Rightarrow B,\ \neg B \vdash \neg A$$
$$\mathtt{A} \to \mathtt{B},\ \mathtt{B} \to \mathtt{0} \vdash \mathtt{A} \to \mathtt{0}$$

Proof. This is just an instance of function composition (Theorem 10).

Since in constructive logic we cannot add and remove negation signs as freely as we can in classical logic, the rule of Contraposition splits into two rules, depending upon whether the consequent of the conditional is a negation or not.

Theorem 17 ('Positive' Contraposition of $\Rightarrow$)**.**

$$A \Rightarrow B \vdash \neg B \Rightarrow \neg A$$
$$\mathtt{A} \to \mathtt{B} \vdash (\mathtt{B} \to \mathtt{0}) \to (\mathtt{A} \to \mathtt{0})$$

Proof. This follows by an application of the Deduction theorem to *modus tollens*.

Theorem 18 ('Negative' Contraposition of $\Rightarrow$)**.**

$$A \Rightarrow \neg B \vdash B \Rightarrow \neg A$$
$$\mathtt{A} \to (\mathtt{B} \to \mathtt{0}) \vdash \mathtt{B} \to (\mathtt{A} \to \mathtt{0})$$

Proof. This is an instance of swapping function arguments (Theorem 13).

Theorem 19 (Contraposition of $\vdash$)**.**

$$\frac{A \vdash B}{\neg B \vdash \neg A} \qquad \frac{\mathtt{A} \vdash \mathtt{B}}{\bar{\mathtt{b}} : \mathtt{B} \to \mathtt{0} \vdash \bar{\mathtt{a}} : \mathtt{A} \to \mathtt{0}}$$

Proof. By the Deduction theorem

$$\frac{\mathtt{A} \vdash \mathtt{B}}{\vdash \mathtt{A} \to \mathtt{B}}$$

and composing this function with $\bar{\mathtt{b}}$ (i.e. by applying the rule of *modus tollens*) we get a function $\bar{\mathtt{a}} : \mathtt{A} \to \mathtt{0}$.

The Law of Excluded Middle does not hold as a rule in constructive logic, and so $\vdash \mathtt{P} + (\mathtt{P} \to \mathtt{0})$ cannot be derived. However, this does *not* mean that we actively *deny* the validity of that rule – we do not assert of any particular proposition that $\neg(P \vee \neg P)$ holds of it. Indeed, we can prove that it is *contradictory* to do so. (In this proof we write $\neg\mathtt{P}$ for $\mathtt{P} \to \mathtt{0}$.)

Theorem 20 (Double Negation of LEM)**.**

$$\vdash \neg\neg(P \vee \neg P)$$
$$\vdash \big((\mathtt{P} + \neg\mathtt{P}) \to \mathtt{0}\big) \to \mathtt{0}$$

Proof. By the Deduction theorem,

$$\frac{(\mathtt{P} + \neg\mathtt{P}) \to \mathtt{0} \;\vdash\; \mathtt{0}}{\vdash\; \big((\mathtt{P} + \neg\mathtt{P}) \to \mathtt{0}\big) \to \mathtt{0}}$$

So all that remains is to prove the top line. From a function $\mathtt{f} : (\mathtt{P} + \neg\mathtt{P}) \to \mathtt{0}$ we can define two functions $\bar{\mathtt{p}} :\equiv \mathtt{f} \circ \mathtt{inl} : \mathtt{P} \to \mathtt{0}$ and $\bar{\bar{\mathtt{p}}} :\equiv \mathtt{f} \circ \mathtt{inr} : \neg\mathtt{P} \to \mathtt{0}$. We then apply $\bar{\bar{\mathtt{p}}}$ to $\bar{\mathtt{p}}$ to get $\bar{\bar{\mathtt{p}}}(\bar{\mathtt{p}}) : \mathtt{0}$. ∎

Thus while constructive logic does not adopt LEM as a general law of logic, it also cannot *deny* that it holds, since this leads immediately to contradiction. Thus we are free to posit as an assumption that some given proposition satisfies LEM (i.e. that we have a token of $(\mathtt{P} + \neg\mathtt{P})$ for the corresponding type $\mathtt{P}$). This is the sense in which classical logic can be recovered as needed, as discussed in Section 1.7.

5.6 The Law of Explosion and Disjunctive Syllogism

The Law of Explosion says that any conclusion follows from contradictory premises. Recall from Section 4.3 that for any type $\mathtt{C}$ we have a function $!_{\mathtt{C}} : \mathtt{0} \to \mathtt{C}$.

Theorem 21 (Explosion)**.**

$$P \,\&\, \neg P \vdash A$$
$$(\mathtt{p}, \bar{\mathtt{p}}) : \mathtt{P} \times (\mathtt{P} \to \mathtt{0}) \vdash \mathtt{A}$$

Proof. By the Principle of Non-Contradiction (Theorem 15) we get $\bar{\mathtt{p}}(\mathtt{p}) : \mathtt{0}$. So we have $!_{\mathtt{A}}(\bar{\mathtt{p}}(\mathtt{p})) : \mathtt{A}$.

Theorem 22 (Disjunctive Syllogism)**.**

$$A \vee B,\ \neg A \vdash B$$
$$\mathtt{A} + \mathtt{B},\ \bar{\mathtt{a}} : \mathtt{A} \to \mathtt{0} \vdash \mathtt{B}$$

Proof. We proceed by case analysis on $\mathtt{A} + \mathtt{B}$. If we have $\langle \mathtt{a}]$ for some $\mathtt{a} : \mathtt{A}$ then $(!_{\mathtt{B}} \circ \bar{\mathtt{a}})(\mathtt{a})$ gives a token of $\mathtt{B}$. If we have $[\mathtt{b}\rangle$ for some $\mathtt{b} : \mathtt{B}$ then we have a token of $\mathtt{B}$ as required. Thus either way we can return a token of $\mathtt{B}$.

5.7 The Material Conditional

In Section 3.6 we sketched a proof that $\to$ satisfies all four lines of the truth table for classical material conditional:

A	B	$A \Rightarrow B$
F	F	T
F	T	T
T	F	F
T	T	T

Two properties evident from this table – that $A \Rightarrow B$ is true whenever A is false, and $A \Rightarrow B$ is true whenever B is true – are called the **paradoxes of material implication**, since they do not fit with the most intuitive understanding of $\Rightarrow$ as an 'if ... then ...' relation. That is, we can have $A \Rightarrow B$ even when there is no connection at all between the ideas expressed by the propositions A and B. The proofs of these principles are both straightforward:

Theorem 23 ($A \Rightarrow B$ with false A)**.**

$$\neg A \vdash A \Rightarrow B$$
$$\bar{\mathtt{a}} : \mathtt{A} \to \mathtt{0} \vdash \mathtt{f} : \mathtt{A} \to \mathtt{B}$$

Proof. Using $!_{\mathtt{B}} : \mathtt{0} \to \mathtt{B}$, we define $\mathtt{f} :\equiv !_{\mathtt{B}} \circ \bar{\mathtt{a}}$.

Theorem 24 ($A \Rightarrow B$ with true B)**.**

$$B \vdash A \Rightarrow B$$
$$\mathtt{b} : \mathtt{B} \vdash \mathtt{k_b} : \mathtt{A} \to \mathtt{B}$$

Proof. Since we are given $\mathtt{b} : \mathtt{B}$ we can define the constant function $\mathtt{k_b} : \mathtt{A} \to \mathtt{B}$ that sends every $\mathtt{a} : \mathtt{A}$ to $\mathtt{b}$.

We can also give a more detailed proof that the third line of the truth table holds:

Theorem 25.

$$A,\ \neg B \ \vdash\ \neg(A \Rightarrow B)$$
$$\mathtt{a} : \mathtt{A},\ \bar{\mathtt{b}} : \mathtt{B} \to \mathtt{0} \ \vdash\ \bar{\mathtt{f}} : (\mathtt{A} \to \mathtt{B}) \to \mathtt{0}$$

Proof. Given any $\mathtt{a} : \mathtt{A}$ and $\bar{\mathtt{b}} : \mathtt{B} \to \mathtt{0}$, the function $\bar{\mathtt{f}}$ applies its input function $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ to $\mathtt{a}$ to produce $\mathtt{f(a)} : \mathtt{B}$, and then applies $\bar{\mathtt{b}}$ to this to get a token of $\mathtt{0}$ as required. We therefore define $\bar{\mathtt{f}} :\equiv [\mathtt{f} \mapsto \bar{\mathtt{b}}(\mathtt{f(a)})]$. ∎

In Section 5.9 we show that although $\to$ behaves like the material conditional in this regard, in constructive logic it is not equivalent to the two classical reformulations $(\neg A) \vee B$ and $\neg(A \,\&\, \neg B)$.

5.8 The Equivalence of DNE and LEM

Neither Double Negation Elimination nor the Law of Excluded Middle hold as rules in constructive logic. We can therefore sometimes prove that certain constructions are impossible by showing that *if* they were possible they would enable us to prove the constructive validity of these rules. In general we don't produce *counterexamples* to invalid rules, as we do in classical logic,[45] so these proofs are the closest we will come to showing that an inference is not constructively valid.

[45] It is possible sometimes to produce models using **Heyting algebras** that play the role of counterexamples for constructive logic, but we will not explore that here.

We can write these 'constructive counterexamples' using the relative constructability notation introduced in Section 5.1: we show $\mathtt{A} \not\vdash \mathtt{B}$ by proving

$$\frac{\mathtt{A} \vdash \mathtt{B}}{(\mathtt{P} \to \mathtt{0}) \to \mathtt{0} \vdash \mathtt{P}} \qquad \text{or} \qquad \frac{\mathtt{A} \vdash \mathtt{B}}{\vdash \mathtt{P} + (\mathtt{P} \to \mathtt{0})}$$

We might now wonder whether a constructive counterexample that results in DNE is different from one that ends in LEM – they both show that the inference is not constructively valid, but do they do so in essentially different ways? It is easy to show that they do not because DNE and LEM can be proved from each other.

Theorem 26 (DNE ⇒ LEM)**.**

$$\frac{(\mathtt{Q} \to \mathtt{0}) \to \mathtt{0} \vdash \mathtt{Q}}{\vdash \mathtt{P} + (\mathtt{P} \to \mathtt{0})}$$

Proof. Recall from Theorem 20 that we can constructively prove the Double Negation of LEM, $\neg\neg(P \vee \neg P)$. Thus by Double Negation Elimination we obtain LEM itself.

Theorem 27 (LEM ⇒ DNE)**.**

$$\frac{\vdash \mathtt{P} + (\mathtt{P} \to \mathtt{0})}{(\mathtt{Q} \to \mathtt{0}) \to \mathtt{0} \vdash \mathtt{Q}}$$

Proof. If LEM holds then in particular we have $\vdash \mathtt{Q} + (\mathtt{Q} \to \mathtt{0})$. That is, from no premises we can construct something that is either a token of $\mathtt{Q}$ or a token of $\mathtt{Q} \to \mathtt{0}$. We then reason by case analysis: we either have a token of $\mathtt{Q}$ already, in which case there is nothing more to prove; or we have a token of $(\mathtt{Q} \to \mathtt{0})$ in which case we apply $(\mathtt{Q} \to \mathtt{0}) \to \mathtt{0}$ to get a contradiction, from which we can derive a token of $\mathtt{Q}$ by Explosion.

In the subsequent sections we explore some further differences between classical and constructive logic. As in Section 5.5 we sometimes write $\neg\mathtt{P}$ for $\mathtt{P} \to \mathtt{0}$ where this is convenient.

5.9 The Relationship between $A \Rightarrow B$, $(\neg A) \vee B$, and $\neg(A \,\&\, \neg B)$

Classically $A \Rightarrow B$ can be defined in terms of $\&$, $\vee$, and $\neg$ in two different but equivalent ways, as $(\neg A) \vee B$, and $\neg(A \,\&\, \neg B)$. In this section we examine the relationship between these three propositions.

Theorem 28.

$$(\neg A) \vee B \vdash A \Rightarrow B$$
$$\neg\mathtt{A} + \mathtt{B} \vdash \mathtt{A} \to \mathtt{B}$$

Proof. By case analysis: we either have a function $\bar{\mathtt{a}} : \neg\mathtt{A}$ or a token $\mathtt{b} : \mathtt{B}$, and we have seen in Theorems 23 and 24 that we can construct a token of $\mathtt{A} \to \mathtt{B}$ from either of these.

Theorem 29.

$$A \Rightarrow B \vdash \neg(A \,\&\, \neg B)$$
$$\mathtt{A} \to \mathtt{B} \vdash (\mathtt{A} \times \neg\mathtt{B}) \to \mathtt{0}$$

Proof. This is the 'negative' contraposition (Theorem 18) of Theorem 25.

These two theorems show that

$$\neg\mathtt{A} + \mathtt{B} \;\vdash\; \mathtt{A} \to \mathtt{B} \;\vdash\; (\mathtt{A} \times \neg\mathtt{B}) \to \mathtt{0}$$

However, the reverse constructions *cannot* be carried out constructively: if they could, then the constructive validity of Double Negation Elimination and the Law of Excluded Middle would follow, which we know is not the case.

Theorem 30.

$$\frac{\mathtt{A} \to \mathtt{B} \;\vdash\; \neg\mathtt{A} + \mathtt{B}}{\vdash \neg\mathtt{A} + \mathtt{A}}$$

Proof. Given $\mathtt{A} \to \mathtt{B} \vdash \neg\mathtt{A} + \mathtt{B}$ for arbitrary $\mathtt{A}$ and $\mathtt{B}$ then we have in particular $\mathtt{A} \to \mathtt{A} \vdash \neg\mathtt{A} + \mathtt{A}$. But we always have the identity function $\mathtt{id}_{\mathtt{A}} : \mathtt{A} \to \mathtt{A}$, so we require no further premises in order to define it. Thus we obtain LEM.

Theorem 31.

$$\frac{(\mathtt{A} \times \neg\mathtt{B}) \to \mathtt{0} \;\vdash\; \mathtt{A} \to \mathtt{B}}{\neg\mathtt{B} \to \mathtt{0} \;\vdash\; \mathtt{B}}$$

Proof. Given $(\mathtt{A} \times \neg\mathtt{B}) \to \mathtt{0} \vdash \mathtt{A} \to \mathtt{B}$ for arbitrary $\mathtt{A}$ and $\mathtt{B}$ then we have in particular $(\mathtt{1} \times \neg\mathtt{B}) \to \mathtt{0} \vdash \mathtt{1} \to \mathtt{B}$. But it is clear that $\neg\mathtt{B} \to \mathtt{0} \vdash (\mathtt{1} \times \neg\mathtt{B}) \to \mathtt{0}$ and $\mathtt{1} \to \mathtt{B} \vdash \mathtt{B}$. Putting these together gives Double Negation Elimination.

So constructively these three propositions are strictly ordered in strength: the strongest is $\neg\mathtt{A} + \mathtt{B}$, then $\mathtt{A} \to \mathtt{B}$ is intermediate in strength, and $(\mathtt{A} \times \neg\mathtt{B}) \to \mathtt{0}$ is strictly weaker than the other two.

5.10 Double Negation Introduction and Triple Negation

Although in constructive logic we don't have the rule of Double Negation Elimination (DNE), we can prove Double Negation Introduction (DNI) as a theorem:

Theorem 32.

$$A \vdash \neg\neg A$$
$$\mathtt{A} \vdash (\mathtt{A} \to \mathtt{0}) \to \mathtt{0}$$

Proof. This follows immediately from the Principle of Non-Contradiction using Cut and Currying.

The function $\bar{\bar{\mathtt{a}}} : (\mathtt{A} \to \mathtt{0}) \to \mathtt{0}$ that we obtain from a given $\mathtt{a} : \mathtt{A}$ could be called `evaluate-at-a`, since we define $\bar{\bar{\mathtt{a}}}(\bar{\mathtt{a}}) :\equiv \bar{\mathtt{a}}(\mathtt{a})$.[46]

Although we cannot eliminate double negations *in general*, we can eliminate them when they occur directly in front of negated propositions.

Theorem 33.

$$\neg\neg\neg A \vdash \neg A$$
$$\mathtt{f} : \big((\mathtt{A} \to \mathtt{0}) \to \mathtt{0}\big) \to \mathtt{0} \vdash \mathtt{g} : \mathtt{A} \to \mathtt{0}$$

Proof. By the Deduction theorem,

$$\frac{\big((\mathtt{A} \to \mathtt{0}) \to \mathtt{0}\big) \to \mathtt{0},\ \mathtt{A} \vdash \mathtt{0}}{\big((\mathtt{A} \to \mathtt{0}) \to \mathtt{0}\big) \to \mathtt{0} \vdash \mathtt{A} \to \mathtt{0}}$$

To prove the top line, from $\mathtt{a} : \mathtt{A}$ we construct $\bar{\bar{\mathtt{a}}} : (\mathtt{A} \to \mathtt{0}) \to \mathtt{0}$ as in Theorem 32. Thus we have $(\mathtt{A} \to \mathtt{0}) \to \mathtt{0}$ and $\big((\mathtt{A} \to \mathtt{0}) \to \mathtt{0}\big) \to \mathtt{0}$, and so by *modus ponens* we have the required contradiction.

5.11 de Morgan's laws

In this section we illustrate a substantial difference between classical and constructive logic by showing that the (classically valid) de Morgan laws are not all valid constructively. de Morgan's laws relate certain expressions involving $\&$, $\vee$, and $\neg$. Working constructively means we have to be more careful with negations, and so we have more relationships to check than we would in classical logic:

$$\begin{aligned}
(A \,\&\, B) \quad &\dashv\vdash \neg(\neg A \vee \neg B) \\
(\neg A \,\&\, \neg B) \quad &\dashv\vdash \neg(A \vee B) \\
\neg(A \,\&\, B) \quad &\dashv\vdash (\neg A \vee \neg B) \\
\neg(\neg A \,\&\, \neg B) \quad &\dashv\vdash (A \vee B)
\end{aligned}$$

We must check all eight of these relationships to see which hold constructively and which do not. In each case we write the classical statement (for which the

[46] This is directly analogous with the relationship between a vector space V and its double-dual vector space V^{**}. Recall that for any vector space V the dual space V^* consists of linear functions $f : V \to \mathbb{K}$ (where $\mathbb{K}$ is the field of scalars over which V is defined). Thus for any vector $v \in V$ there is an element $\hat{v} \in V^{**}$ (i.e. a linear function $\hat{v} : V^* \to \mathbb{K}$) that evaluates its input function at v, defined by $\hat{v}(f) :\equiv f(v)$.

entailments hold in both directions) followed by the constructive version (for which, in some cases, only one direction of entailment holds).

5.11.1 First de Morgan Law

Theorem 34 (First de Morgan Law)**.**

$$(A \,\&\, B) \dashv\vdash \neg(\neg A \vee \neg B)$$
$$(\mathtt{a}, \mathtt{b}) : (\mathtt{A} \times \mathtt{B}) \not\dashv\vdash \mathtt{g} : \big(\neg\mathtt{A} + \neg\mathtt{B}\big) \to \mathtt{0}$$

Proof. $\vdash$: Given $\mathtt{a} : \mathtt{A}$ and $\mathtt{b} : \mathtt{B}$ we can define the function $\mathtt{g}$ by case analysis on its input: if its input is a token of $\neg\mathtt{A}$ then apply this to $\mathtt{a}$, whereas if its input is a token of $\neg\mathtt{B}$ then apply this to $\mathtt{b}$.

$\not\dashv$: This direction is not constructively valid: if we could carry out this construction then we could use it to prove that DNE is constructively valid:

Lemma 1.

$$\frac{\big(\neg\mathtt{A} + \neg\mathtt{B}\big) \to \mathtt{0} \;\vdash\; \mathtt{A} \times \mathtt{B}}{\neg\mathtt{A} \to \mathtt{0} \;\vdash\; \mathtt{A}}$$

Proof. If the top line holds for arbitrary types $\mathtt{A}$ and $\mathtt{B}$ then in particular we have

$$\big(\neg\mathtt{A} + \neg\mathtt{1}\big) \to \mathtt{0} \;\vdash\; \mathtt{A} \times \mathtt{1}$$

Since (trivially) $(\mathtt{A} \to \mathtt{0}) \to \mathtt{0} \vdash \big(\neg\mathtt{A} + \neg\mathtt{1}\big) \to \mathtt{0}$ and $\mathtt{A} \times \mathtt{1} \vdash \mathtt{A}$, this gives DNE.

5.11.2 Second de Morgan Law

Theorem 35 (Second de Morgan Law)**.**

$$(\neg A \,\&\, \neg B) \dashv\vdash \neg(A \vee B)$$
$$(\bar{\mathtt{a}}, \bar{\mathtt{b}}) : \big(\neg\mathtt{A} \times \neg\mathtt{B}\big) \dashv\vdash \mathtt{g} : (\mathtt{A} + \mathtt{B}) \to \mathtt{0}$$

Proof. $\vdash$: Given $\bar{\mathtt{a}} : \neg\mathtt{A}$ and $\bar{\mathtt{b}} : \neg\mathtt{B}$, we define the function $\mathtt{g}$ by case analysis: it applies either $\bar{\mathtt{a}}$ or $\bar{\mathtt{b}}$ to its input.

$\dashv$: Given a function $\mathtt{g} : (\mathtt{A}+\mathtt{B}) \to \mathtt{0}$, we define $\bar{\mathtt{a}} :\equiv \mathtt{g} \circ \mathtt{inl} : \neg\mathtt{A}$ and $\bar{\mathtt{b}} :\equiv \mathtt{g} \circ \mathtt{inr} : \neg\mathtt{B}$. (Note that this is a generalisation of the technique used in Theorem 20.)

5.11.3 Third de Morgan Law

Theorem 36 (Third de Morgan Law)**.**

$$\neg(A \,\&\, B) \dashv\vdash (\neg A \vee \neg B)$$
$$\mathtt{f} : (\mathtt{A} \times \mathtt{B}) \to 0 \dashv\nvdash \big(\neg\mathtt{A} + \neg\mathtt{B}\big)$$

Proof. $\nvdash$: This direction is not constructively valid. The function $\mathtt{f}$ can only be applied to pairs $(\mathtt{a}, \mathtt{b})$. From this, we cannot construct a function $\neg\mathtt{A}$, since we have no way of creating an element of an arbitrary type $\mathtt{B}$ given only a token of $\mathtt{A}$ as input. Likewise, we can't construct a function $\neg\mathtt{B}$. Thus we cannot construct a token of the coproduct $\big(\neg\mathtt{A} + \neg\mathtt{B}\big)$.

$\dashv$: By case analysis we either have a token $\bar{\mathtt{a}} : \neg\mathtt{A}$ or a token $\bar{\mathtt{b}} : \neg\mathtt{B}$, and either of these is sufficient to construct a function $(\mathtt{A} \times \mathtt{B}) \to 0$. If we have $\bar{\mathtt{a}}$ we define $\mathtt{f} :\equiv \bar{\mathtt{a}} \circ \mathtt{pr}_1$, whereas if we have $\bar{\mathtt{b}}$ we define $\mathtt{f} :\equiv \bar{\mathtt{b}} \circ \mathtt{pr}_2$.

5.11.4 Fourth de Morgan Law

Theorem 37 (Fourth de Morgan Law)**.**

$$\neg(\neg A \,\&\, \neg B) \;\dashv\vdash\; (A \vee B)$$
$$\mathtt{f} : \big(\neg\mathtt{A} \times \neg\mathtt{B}\big) \to 0 \;\dashv\nvdash\; \mathtt{A} + \mathtt{B}$$

Proof. $\nvdash$: This direction is not constructively valid: if we could carry out this construction then we could use it to prove that DNE is constructively valid:

Lemma 2.

$$\frac{\big(\neg\mathtt{A} \times \neg\mathtt{B}\big) \to 0 \;\vdash\; \mathtt{A} + \mathtt{B}}{(\mathtt{A} \to 0) \to 0 \;\vdash\; \mathtt{A}}$$

Proof. If the top line holds for arbitrary types $\mathtt{A}$ and $\mathtt{B}$ then in particular we have:

$$\big(\neg\mathtt{A} \times \neg 0\big) \to 0 \;\vdash\; \mathtt{A} + 0$$

Since (trivially) $(\mathtt{A} \to 0) \to 0 \vdash \big(\neg\mathtt{A} \times \neg 0\big) \to 0$ and $\mathtt{A} + 0 \vdash \mathtt{A}$ this gives DNE. ∎

$\dashv$: By case analysis: if we have a token $\mathtt{a} : \mathtt{A}$ then we define $\mathtt{f}((\bar{\mathtt{a}}, \bar{\mathtt{b}})) :\equiv \bar{\mathtt{a}}(\mathtt{a})$ whereas if we have a token $\mathtt{b} : \mathtt{B}$ then we define $\mathtt{f}((\bar{\mathtt{a}}, \bar{\mathtt{b}})) :\equiv \bar{\mathtt{b}}(\mathtt{b})$. ∎

5.11.5 Summary of the constructive de Morgan laws

The classical de Morgan laws can be broken up into eight different entailments, and we have shown above that only five of them hold constructively:

$$\begin{array}{ll} (\mathtt{A} \times \mathtt{B}) & \not\dashv \vdash (\neg\mathtt{A} + \neg\mathtt{B}) \to \mathtt{0} \\ (\neg\mathtt{A} \times \neg\mathtt{B}) & \dashv \vdash \ (\mathtt{A} + \mathtt{B}) \to \mathtt{0} \\ (\mathtt{A} \times \mathtt{B}) \to \mathtt{0} & \dashv \not\vdash (\neg\mathtt{A} + \neg\mathtt{B}) \\ (\neg\mathtt{A} \times \neg\mathtt{B}) \to \mathtt{0} & \dashv \not\vdash \ (\mathtt{A} + \mathtt{B}) \end{array}$$

Of the three that are not constructively valid we have explicit counterexamples to two of them, in the sense that we can show that if they were constructively valid then we would obtain constructive DNE as a special case. However, for the other constructively invalid law, namely

$$(\mathtt{A} \times \mathtt{B}) \to \mathtt{0} \ \not\vdash \ (\neg\mathtt{A} + \neg\mathtt{B})$$

we do not have such an argument. To demonstrate that this is not constructively valid we need the more powerful technique of *Heyting algebra models*, which are beyond the scope of this Primer.

6 Quantifiers

In Section 4 we demonstrated how constructive propositional logic can be incorporated into type theory. But propositional logic isn't enough to do mathematics – we need predicate logic as well. We therefore need to find a way of defining *universal* and *existential quantifiers*. In this section we show how to do that, first examining how the quantifiers should be understood in constructive logic under the BHK interpretation, and then defining how they are implemented in the type theory. We must therefore work out what a certificate to a quantified proposition is.

A quantified proposition is of the form $(\forall x)\,\phi(x)$ or $(\exists x)\,\phi(x)$ for some **predicate** ϕ, with x ranging over some 'domain of discourse' (everything in the domain satisfies the predicate or something in the domain satisfies the predicate respectively). Sometimes we make the domain of discourse explicit. In set theory the domain of discourse is some set A, so we would write $(\forall x \in A)\,\phi(x)$ or $(\exists x \in A)\,\phi(x)$.

To translate this into type theory we first need to translate the predicate ϕ. ϕ itself is not a proposition, but rather corresponds to something like '... is even' or '... is prime', where the '...' has not been filled in with any particular entity.[47] So it is more like a function that takes some entity x as input and returns as output a proposition involving x. For example, if ϕ corresponds to '... is prime', then $\phi(5)$ is the proposition '5 is prime' and $\phi(12)$ is the proposition '12 is prime'.

But what kind of function could this be? A function of type $\mathtt{A} \to \mathtt{B}$ takes a token of $\mathtt{A}$ as input and returns a token of $\mathtt{B}$. However, we want a predicate to be a function that takes a token of some type as input and returns a proposition (i.e. a *type*) as output. What could the type of such a function be?

6.1 A new type called TYPE

To define functions that return propositions as output, we must introduce a new type that has as its *tokens* all the types considered so far. That is, for each $\mathtt{A}$, $\mathtt{B}$, $\mathtt{X} \times \mathtt{Y}$, etc. we have $\mathtt{A} : \mathtt{TYPE}$, $\mathtt{B} : \mathtt{TYPE}$, $\mathtt{X} \times \mathtt{Y} : \mathtt{TYPE}$, etc. Although it might appear a little odd and perhaps a little dangerous at first, this does seem to solve our problem: a predicate $\mathtt{P}$ that asserts things about tokens of type $\mathtt{A}$ is a function of type $\mathtt{A} \to \mathtt{TYPE}$. It takes a token $\mathtt{x} : \mathtt{A}$ as input, and returns $\mathtt{P(x)} : \mathtt{TYPE}$ as output.

[47] Cf. Frege on the "unsaturated" nature of functions (see for example Section 2.4 of `http://plato.stanford.edu/entries/frege/`).

However, to accept this solution we need to overcome two difficulties:

1. Doesn't introducing this 'type of types' immediately open the door to paradox?
2. Isn't it incompatible with the whole idea of having tokens and types as distinct things?

We address the second issue first.

We first introduced types as 'kinds of things that a mathematical entity could be'. Some mathematical entities are integers, so *integer* is a type; some mathematical entities are points in the Euclidean plane, so *point in the Euclidean plane* is a type; and so on. We put no kind of limit on what types there are, so any well-defined 'kind of mathematical entity' corresponds to a type.

If we are to take this defining idea seriously, we have two options: we can either

(i) *deny* that types themselves are mathematical entities, in which case *type* is not a kind of thing that mathematical entities could be; or

(ii) deny the assumption that for every kind of mathematical entity there is a type to which they belong.

The former option seems unmotivated: we're doing mathematics and logic using types, so they certainly seem like mathematical entities of some kind. The second option is somewhat less unmotivated, but needs to be spelled out in more detail. Which kinds of mathematical entities *don't* have types to which they belong? Is it just types themselves – and if so, why?

The main motivation for making such a distinction is the worry that there is the potential for introducing paradox or circularity. Introducing `TYPE` seems dangerous because a type that contains all types as tokens must contain itself as a token, which surely leads to trouble. And indeed, if we had $\texttt{TYPE} : \texttt{TYPE}$ then a paradox would arise.[48] But note that the proposal, as stated above, is to introduce 'a type that has as its tokens all the types considered so far'. To be more precise, the tokens of `TYPE` are all the types whose definitions can be given *without introducing* `TYPE` *itself*. Thus, for example, the Zero and Unit types are tokens of `TYPE`, and if $\texttt{A} : \texttt{TYPE}$ and $\texttt{B} : \texttt{TYPE}$ then $\texttt{A} \times \texttt{B} : \texttt{TYPE}$, $\texttt{A} + \texttt{B} : \texttt{TYPE}$, and $\texttt{A} \to \texttt{B} : \texttt{TYPE}$. However, `TYPE` itself cannot be a token of `TYPE` and nor is any predicate type $\texttt{A} \to \texttt{TYPE}$, since these could not have been defined before we introduced `TYPE` itself. By analogy

[48] See Section 6 of `http://plato.stanford.edu/entries/type-theory/`

with the terminology of category theory, we may call the tokens of `TYPE` 'small types'.

Now, we can repeat the same argument at the next level up. Just as we argued that there should be a type called `TYPE` having small types as its tokens, shouldn't there also be a type that has `TYPE` and the predicates `A → TYPE` (and any other types that we can now define involving `TYPE`) as its tokens as well? The answer of course is yes, and so we cannot rest at introducing just `TYPE` alone. To do justice to the idea that *types are tokens of some higher type* we must introduce a whole hierarchy $\texttt{TYPE}_0$, $\texttt{TYPE}_1$, $\texttt{TYPE}_2$, ...[49]

However, this becomes a little complicated, and is not really necessary in order to solve the problem that we introduced `TYPE` to solve, namely, to define predicates. So for now we just use `TYPE`, but use it carefully (in particular bearing in mind that `TYPE` is something different from the types it contains – we don't have `TYPE : TYPE`). So *strictly speaking* what we say in the following involving `TYPE` is not quite correct since it ignores or collapses the hierarchy of universes, but it can be made rigorous.

6.2 The BHK interpretation of quantifiers

In this section we consider quantified statements in set theoretic notation, i.e. $(\forall x \in A)\, \phi(x)$ and $(\exists x \in A)\, \phi(x)$, and how to understand them under the BHK interpretation.

Given a predicate ϕ, the **universally quantified** proposition $(\forall x \in A)\, \phi(x)$ says of every element $x \in A$ that ϕ holds of it, i.e. that the proposition $\phi(x)$ is true.

Under the BHK interpretation we can only assert that $\phi(x)$ is true when we have a certificate of it, and so we can only claim that $\phi(x)$ is true for all $x \in A$ if we can produce a certificate for each such proposition. Thus, under the BHK interpretation, a certificate to $(\forall x \in A)\, \phi(x)$ must be a function that produces a certificate to $\phi(x)$ for any $x \in A$ given as input.

Given a predicate ϕ, the **existentially quantified** proposition $(\exists x \in A)\, \phi(x)$ says that there is at least one element $x \in A$ such that ϕ holds of it, i.e. such that the proposition $\phi(x)$ is true.

Again, we can only claim that $\phi(x)$ is true for a given x if we have a certificate to that proposition. So we interpret the existentially qualified proposition as saying that there is at least one x for which there is a certificate to $\phi(x)$. But again, it is

[49] Then what we've called `TYPE` above is really $\texttt{TYPE}_0$, and any type whose definition depends upon $\texttt{TYPE}_i$ is then a token of $\texttt{TYPE}_{i+1}$. To handle this hierarchy we invoke **Grothendieck universes**, and accordingly in the HoTT Book, $\texttt{TYPE}_i$ is written $\mathcal{U}_i$. For more details on this see the HoTT Book, Section 1.3 and p. 55.

not enough to claim that such an x and such a certificate exists – we must be able to produce them. Thus, under the BHK interpretation, a certificate to $(\exists x \in A)\, \phi(x)$ consists of a particular $x \in A$, along with a certificate to the proposition $\phi(x)$.

6.3 Translating the BHK interpretation of quantifiers into the type theory

Recall a predicate ϕ corresponds to a function $\mathtt{P} : \mathtt{A} \to \mathtt{TYPE}$ that takes a token $\mathtt{x} : \mathtt{A}$ as input and returns as output the type $\mathtt{P}(\mathtt{x})$ corresponding to the proposition $\phi(x)$. Now we put this together with the BHK interpretation of quantified propositions to show how to express them in type theory.

6.3.1 Dependent Function Types

Corresponding to the universally quantified proposition $(\forall x \in A)\, \phi(x)$ we require a type whose tokens are functions that take a token $\mathtt{x} : \mathtt{A}$ as input and return as output a token of type $\mathtt{P}(\mathtt{x})$.

Until now we have only seen functions of types like $\mathtt{A} \to \mathtt{B}$ that return outputs of a given *fixed* type $\mathtt{B}$, whereas the functions we require here give outputs whose type is different depending upon which token of the input type they are given. We therefore need to introduce a new way of forming types, alongside the product, coproduct and function types defined in previous sections. This is called the **dependent function type**.

The dependent function type has one type former, which takes as input a type $\mathtt{A}$ and a predicate on $\mathtt{A}$, i.e. a function $\mathtt{P} : \mathtt{A} \to \mathtt{TYPE}$. We write the resulting dependent function type as

$$\prod_{\mathtt{x}:\mathtt{A}} \mathtt{P}(\mathtt{x}) \qquad \text{or} \qquad \langle \mathtt{x} : \mathtt{A} \rangle \to \mathtt{P}(\mathtt{x})$$

The former notation recalls the set-theoretic notation for dependent products and resembles the standard notation for universal quantification, and the latter resembles the notation for (non-dependent) functions. Since we write the dependent function type with the Π symbol, it is often referred to as the Π-type.[50]

There is one token constructor for the dependent function type, which is a straightforward generalisation of the λ-abstraction that we use for function types (see Section 2.4). The only difference is that whereas λ-abstraction for function types takes an expression Φ of type $\mathtt{B}$ that involves a variable $\mathtt{x}$ of type $\mathtt{A}$, for dependent

[50] Alternatively it is also called the **dependent product type**, but we avoid this terminology.

function types we require an expression whose type $\mathtt{P(x)}$ depends upon $\mathtt{x}$. As with non-dependent functions, we write the function as $[\mathtt{x} \mapsto \Phi]$.

Application of a dependent function is just the same as application of a non-dependent function: when we apply $\mathtt{f} : \langle \mathtt{x} : \mathtt{A} \rangle \to \mathtt{P(x)}$ to any $\mathtt{a} : \mathtt{A}$ we get some $\mathtt{f(a)} : \mathtt{P(a)}$.

As a special case, for any given type $\mathtt{B}$ we can define the 'constant predicate' that assigns to each $\mathtt{x} : \mathtt{A}$ the same type $\mathtt{B}$. When we choose a constant predicate for $\mathtt{P}$ we get a 'dependent function type' that is no longer dependent, which we might write as $\mathtt{f} : \langle \mathtt{x} : \mathtt{A} \rangle \to \mathtt{B}$. A token of this type is a function that, when applied to some $\mathtt{a} : \mathtt{A}$, returns some token $\mathtt{f(a)} : \mathtt{B}$, in other words simply a function $\mathtt{A} \to \mathtt{B}$. So (non-dependent) function types $\mathtt{A} \to \mathtt{B}$ are just special cases of dependent function types – they're ones that have a 'constant predicate' that picks the same type $\mathtt{B}$ for every input token.

6.3.2 Dependent Pair Types

Corresponding to the existentially quantified proposition $(\exists x \in A)\, \phi(x)$ we require a type whose tokens are pairs of the form $(\mathtt{x}, \mathtt{p})$, where $\mathtt{x} : \mathtt{A}$ and $\mathtt{p} : \mathtt{P(x)}$.

This is unlike the pairs that are tokens of the product type $\mathtt{A} \times \mathtt{B}$, since in that case the second component of every pair is of the given fixed type $\mathtt{B}$. The pairs we require now have second components whose type is different depending upon the first component. We therefore need to introduce another new way of forming types, called the **dependent pair type**.

The dependent pair type has one type former, which takes as input a type $\mathtt{A}$ and a predicate on $\mathtt{A}$, i.e. a function $\mathtt{P} : \mathtt{A} \to \mathtt{TYPE}$. We write the resulting dependent pair type as

$$\sum_{\mathtt{x:A}} \mathtt{P(x)} \qquad \text{or} \qquad \langle \mathtt{x} : \mathtt{A} \rangle \times \mathtt{P(x)}$$

The former notation recalls the set-theoretic notation for disjoint sums/coproducts and resembles the standard notation for existential quantification, and the latter resembles the notation for (non-dependent) products. Since we write the dependent pair type with the Σ symbol, it is often referred to as the Σ-type.[51]

The token constructor for the dependent pair type is just like that of the product type, with the simple generalisation that when the first component is $\mathtt{a} : \mathtt{A}$, the type of the second component is $\mathtt{P(a)}$.

If, as in the previous section, we choose $\mathtt{P}$ to be a 'constant predicate' that picks the same type $\mathtt{B}$ for each input $\mathtt{a} : \mathtt{A}$, then the resulting $\sum$-type, which we might

[51] Alternatively it is also called the **dependent sum type**, but we will avoid this terminology.

write as $\langle \mathtt{a} : \mathtt{A} \rangle \times \mathtt{B}$, is again no longer dependent. The tokens of this type are pairs $(\mathtt{a}, \mathtt{b})$, i.e. tokens of the product type $\mathtt{A} \times \mathtt{B}$. So product types $\mathtt{A} \times \mathtt{B}$ are just special cases of dependent pair types – they're ones that have a 'constant predicate' that picks the same type $\mathtt{B}$ for every token of its input.

The elimination rule for dependent pair types is similar to that for product types. Recall from Section 4.1.2 that to define a function $\mathtt{f} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{C}$ we require a function $\mathtt{g} : \mathtt{A} \to (\mathtt{B} \to \mathtt{C})$, so for each $\mathtt{a} : \mathtt{A}$ the output of $\mathtt{g}$ is a function $\mathtt{g}_\mathtt{a} : \mathtt{B} \to \mathtt{C}$ whose input type is $\mathtt{B}$. Uncurrying (Theorem 12) then produces from $\mathtt{g}$ a function $\mathtt{f} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{C}$ as required.

Similarly, to define a function $\mathtt{f} : \big(\langle \mathtt{x} : \mathtt{A} \rangle \times \mathtt{P}(\mathtt{x})\big) \to \mathtt{C}$ we require a dependent function $\mathtt{g} : \langle \mathtt{x} : \mathtt{A} \rangle \to (\mathtt{P}(\mathtt{x}) \to \mathtt{C})$, so for each $\mathtt{a} : \mathtt{A}$ the output of $\mathtt{g}$ is a function $\mathtt{g}_\mathtt{a} : \mathtt{P}(\mathtt{a}) \to \mathtt{C}$ whose input type depends upon the token $\mathtt{a}$. The same Uncurrying procedure then produces from $\mathtt{g}$ a function $\mathtt{f} : \big(\langle \mathtt{x} : \mathtt{A} \rangle \times \mathtt{P}(\mathtt{x})\big) \to \mathtt{C}$ as required.

Uncurrying therefore takes as input a dependent function $\mathtt{g} : \prod_{\mathtt{x}:\mathtt{A}} \mathtt{P}(\mathtt{x}) \to \mathtt{C}$ and returns as output a function $\mathtt{f} : \big(\sum_{\mathtt{x}:\mathtt{A}} \mathtt{P}(\mathtt{x})\big) \to \mathtt{C}$. This corresponds to the logical inference from $\forall x\,(\phi(x) \Rightarrow c)$ to $(\exists x\,\phi(x)) \Rightarrow c$.

6.4 Another interpretation of dependent pair types

We noted in Section 1.3 that there's a temptation to think of types and their tokens as being like sets and their members. While we should continue to resist this temptation, there are similarities. So just temporarily, we set aside the admonition of Section 1.3 and take the analogy fairly literally. On that understanding, the dependent pair type takes on a new interpretation: if $\mathtt{A}$ is like a set of elements then $\sum_{\mathtt{x}:\mathtt{A}} \mathtt{P}(\mathtt{x})$ is (approximately) like the *subset* of $\mathtt{A}$ for which the property corresponding to $\mathtt{P}$ holds, i.e. it's like $\{x \in A \,|\, P(x)\}$. Dependent pair types are therefore a way of constructing the *subtypes* discussed in Section 1.4.

However, the type $\sum_{\mathtt{x}:\mathtt{A}} \mathtt{P}(\mathtt{x})$ is not exactly like a subset, because a token of this type consists of a pair $(\mathtt{x}, \mathtt{p})$, where $\mathtt{x} : \mathtt{A}$ and $\mathtt{p} : \mathtt{P}(\mathtt{x})$. Given a different token $\mathtt{p}' : \mathtt{P}(\mathtt{x})$ for the same $\mathtt{x} : \mathtt{A}$, we can form the pair $(\mathtt{x}, \mathtt{p}')$, which is also a token of $\sum_{\mathtt{x}:\mathtt{A}} \mathtt{P}(\mathtt{x})$. So the same token $\mathtt{x} : \mathtt{A}$ can occur multiple times in multiple pairs – one for each distinct certificate of $\mathtt{P}(\mathtt{x})$. Counted this way, there might be *many more* tokens in $\sum_{\mathtt{x}:\mathtt{A}} \mathtt{P}(\mathtt{x})$ than in $\mathtt{A}$ itself.[52]

[52] One response to the above difference between subtypes and subsets is to restrict the use of the word 'subtype' to the case where all tokens $(\mathtt{a}, \mathtt{p}), (\mathtt{a}, \mathtt{q}), \ldots$ for any given $\mathtt{a} : \mathtt{A}$ are identical. This is what is done in the HoTT Book. However, the analogy between dependent pair types and subsets is sufficiently useful that we think the appropriate response is instead to expand the use of the word 'subtype' to dependent pair types in general and to bear in mind the above caveat, rather than imposing such a restriction. In Part II we revisit subtypes in more detail and see

In these two interpretations of the dependent pair type we see an illustration of a big departure from the usual way of thinking in mathematics. In set theory built on first-order logic, the interpretations of $(\exists x \in A)\,\phi(x)$ and $\{x \in A \,|\, \phi(x)\}$ are completely different: the former is a proposition which is simply either true or false, while the latter is a set, an object that can have substantial content. They're related, of course – non-emptiness of the latter set implies the truth of the former proposition – but conceptually they're quite distinct.

In constructive logic with Propositions as Types, on the other hand, the distinction between the two notions is eliminated. Propositions are only asserted to be true when we construct certificates to their truth, and so the proof of an existential claim requires an *example* of the thing asserted to exist. Thus, since each example is as good a certificate as any other, the idea of 'the collection of all...' and the idea of 'there exists some...' collapse into each other.

6.5 More complex quantification

Having defined predicates as things that assert propositions about the tokens of some type, we should look at *relations*, i.e. things that assert propositions about multiple tokens of multiple types. Just as a predicate on $\mathtt{A}$ is a function of type $\mathtt{A} \to \mathtt{TYPE}$, so a relation is a function from a product type to $\mathtt{TYPE}$, such as $\mathtt{R} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{TYPE}$. The application of relation $\mathtt{R}$ to inputs $\mathtt{a} : \mathtt{A}$ and $\mathtt{b} : \mathtt{B}$ therefore gives a type whose tokens $\mathtt{r} : \mathtt{R}(\mathtt{a}, \mathtt{b})$ are certificates to the fact that $\mathtt{a}$ and $\mathtt{b}$ stand in this relation. As in Section 2.6, we define a function that takes a single input of product type by defining a function of the corresponding type $\mathtt{A} \to (\mathtt{B} \to \mathtt{TYPE})$. The latter allows partial application: applying a relation of type $\mathtt{A} \to (\mathtt{B} \to \mathtt{TYPE})$ to a token $\mathtt{a} : \mathtt{A}$ gives a predicate of type $\mathtt{B} \to \mathtt{TYPE}$ on $\mathtt{B}$.

Given a relation $\mathtt{R} : (\mathtt{A} \times \mathtt{B}) \to \mathtt{TYPE}$ we can form dependent function types and dependent pair types. For example, we can define the dependent function type

$$\prod_{\mathtt{a}:\mathtt{A}} \prod_{\mathtt{b}:\mathtt{B}} \mathtt{R}(\mathtt{a}, \mathtt{b})$$

whose tokens are functions taking tokens $\mathtt{a}$ and $\mathtt{b}$ as input and returning as output a token of $\mathtt{R}(\mathtt{a}, \mathtt{b})$. Thus a token of this dependent function type represents the universally quantified proposition $(\forall \mathtt{a} : \mathtt{A})(\forall \mathtt{b} : \mathtt{B})\,\mathtt{R}(\mathtt{a}, \mathtt{b})$. Similarly, we can form the dependent pair type

$$\sum_{\mathtt{a}:\mathtt{A}} \sum_{\mathtt{b}:\mathtt{B}} \mathtt{R}(\mathtt{a}, \mathtt{b})$$

corresponding to the proposition $(\exists \mathtt{a} : \mathtt{A})(\exists \mathtt{b} : \mathtt{B})\,\mathtt{R}(\mathtt{a}, \mathtt{b})$.

another way of defining them.

The above expressions involve just a single type of quantifier, either universal or existential, but we can also define mixed-quantifier propositions of the form $(\forall \mathtt{a} : \mathtt{A})\,(\exists \mathtt{b} : \mathtt{B})\,\mathtt{R}(\mathtt{a}, \mathtt{b})$ and $(\exists \mathtt{a} : \mathtt{A})\,(\forall \mathtt{b} : \mathtt{B})\,\mathtt{R}(\mathtt{a}, \mathtt{b})$ which correspond respectively to the following dependent types:

$$\prod_{\mathtt{a:A}} \sum_{\mathtt{b:B}} \mathtt{R}(\mathtt{a}, \mathtt{b}) \qquad \text{and} \qquad \sum_{\mathtt{a:A}} \prod_{\mathtt{b:B}} \mathtt{R}(\mathtt{a}, \mathtt{b})$$

where

- A token of $\prod_{\mathtt{a:A}} \sum_{\mathtt{b:B}} \mathtt{R}(\mathtt{a}, \mathtt{b})$ is a function that takes an $\mathtt{x} : \mathtt{A}$ as input and returns as output a pair $(\mathtt{y}, \mathtt{r})$, where $\mathtt{y} : \mathtt{B}$ and $\mathtt{r} : \mathtt{R}(\mathtt{x}, \mathtt{y})$
- A token of $\sum_{\mathtt{a:A}} \prod_{\mathtt{b:B}} \mathtt{R}(\mathtt{a}, \mathtt{b})$ is a pair $(\mathtt{x}, \mathtt{f})$ where $\mathtt{x} : \mathtt{A}$ and $\mathtt{f} : \prod_{\mathtt{b:B}} \mathtt{R}(\mathtt{x}, \mathtt{b})$ is a function that takes a $\mathtt{y} : \mathtt{B}$ and returns a token $\mathtt{r} : \mathtt{R}(\mathtt{x}, \mathtt{y})$.

We can also define relations not just on product types but on dependent pair types as well. This allows us to write quantified propositions in which later types depend upon earlier tokens, such as

$$\prod_{\mathtt{a:A}} \sum_{\mathtt{p:P(a)}} \mathtt{Q}(\mathtt{a}, \mathtt{p})$$

where $\mathtt{P} : \mathtt{A} \to \mathtt{TYPE}$ and $\mathtt{Q} : \big(\langle \mathtt{a} : \mathtt{A} \rangle \times \mathtt{P}(\mathtt{a})\big) \to \mathtt{TYPE}$.

Finally, we can quantify over dependent types as well, for example

$$\sum_{\mathtt{g:}\langle \mathtt{x:X} \rangle \to \mathtt{P(x)}} \prod_{\mathtt{x:X}} \mathtt{S}(\mathtt{x}, \mathtt{g}(\mathtt{x}))$$

6.6 Polymorphism: Quantifying over TYPE

We have given many definitions above that are prefaced (implicitly or explicitly) with something like 'For any type A ...'.[53] This is a kind of informal quantification over types, but it can be made formal. Just as we use dependent function types to quantify over the tokens of a type, translating 'For any $\mathtt{a} : \mathtt{A}$ ...' into $\prod_{\mathtt{a:A}} \ldots$, we can likewise quantify over the tokens of TYPE, which are types.[54] We therefore translate 'For any type A ...' as $\prod_{\mathtt{A:TYPE}} \ldots$.

[53] Strictly then the thing we define should be labelled with the particular choice of type A, but usually we omit it to make the notation more concise.

[54] There are subtleties to do with exactly which types are tokens of TYPE – recall that TYPE cannot contain *all* types as its tokens, on pain of paradox. Just as with the hierarchy of higher types mentioned in Section 6.1, these details need not concern us for now.

So, for example, in Section 5 we took '$\mathtt{A}$', '$\mathtt{B}$', etc. to stand for arbitrary types. We could state more precisely the sense in which each of the constructions works for any types $\mathtt{A}$, $\mathtt{B}$, etc. by the use of quantifiers.

For example, in Section 5 we proved an Absorption rule (Theorem 5):

$$\mathtt{A} + (\mathtt{A} \times \mathtt{B}) \vdash \mathtt{A}$$

by showing that we could define a function of type $\mathtt{A} + (\mathtt{A} \times \mathtt{B}) \to \mathtt{A}$. As with all the proofs in Section 5, '$\mathtt{A}$' and '$\mathtt{B}$' stand for arbitrary types, and so this rule is implicitly quantified over $\mathtt{TYPE}$. To make this quantification formal we can write the Absorption rule as

$$\prod_{\mathtt{X:TYPE}} \prod_{\mathtt{Y:TYPE}} \mathtt{X} + (\mathtt{X} \times \mathtt{Y}) \to \mathtt{X}$$

a token of which is a function taking a particular $\mathtt{A}$ and $\mathtt{B}$ as inputs and returning a certificate $\mathtt{f}_{\mathtt{A,B}} : \mathtt{A} + (\mathtt{A} \times \mathtt{B}) \to \mathtt{A}$ for those particular types.[55]

For simple statements like the one above, we generally leave the quantification over types implicit, and omit the type labels where this can be done without ambiguity. But when we need to be more precise then this explicit quantification over types, called **polymorphism**, is used.

[55] One of the subtleties mentioned above is that, since there is no single type that contains all types as its tokens, such a quantification can never quite capture the idea that, for example, the Absorption rule holds for literally *any* pair of types. Rather, it must be relativised to a particular choice of 'universe' (see footnote 49).

7 Identity Types

The type theory defined in the previous sections, with the product, coproduct, function, Zero and Unit types defined in Section 4 and the dependent types defined in Section 6, is a powerful language for mathematics. However, there is another essential component that remains to be added, namely a way of talking about the *identity* relation.

With the elements and methods of type definition given so far we can, for example, introduce a type that plays the role of the natural numbers $\mathbb{N}$, and we can define all the usual arithmetic functions such as addition and multiplication.[56] But without a way of expressing identity within the theory we can't do even the most basic number theory: we can't write equations, and so we can't even state (let alone prove) that, for example, the sum of the natural numbers up to n is $n(n+1)/2$. Identity is an essential component of a language that claims to serve as a foundation for mathematics.

One thing we can do is introduce a symbol into the language to denote identity between expressions that name types and tokens. If the expressions exp_1 and exp_2 both name types or tokens then the expression $exp_1 \equiv exp_2$ says that these expressions in fact name the same type or token. This is sufficient to allow us to express mathematical facts involving identity and equality, such as the number-theoretic facts mentioned above.[57] However, a statement of identity between two tokens is a proposition, and for any other kind of mathematical proposition that we can express there is a corresponding type. The resources defined above even supplemented as just suggested do not provide a way to form a type corresponding to such propositions. We must therefore supplement the language by introducing *identity types*. Whereas the above approach, introduces 'external' or 'judgmental' identity, and allow us to express identifications, the introduction of 'internal' or 'propositional' identity via identity types allows us to treat identifications as objects of study in their own right. Moreover, since the type theory is constructive and intensional, representing identity internally in the type theory allows identity to take on the rich and interesting structure of **infinity groupoids** that makes HoTT of so much mathematical interest and allows the homotopy interpretation of the type theory. (We explain the beginnings of this structure below but it is discussed in much greater detail in Chapter 2 of the HoTT Book.) In this section we introduce this remaining piece of the language, and explain some subtleties about how it works.[58]

[56] These definitions are given in Part II.

[57] In what follows we talk of 'identity' not 'equality'; the latter is used interchangeably with the former in HoTT.

[58] In what follows we derive the elimination rule for identity types, known as 'path in-

7.1 Type former for the identity type

As stated above, to assert that two things are identical is to assert a proposition, and since types that correspond to such propositions cannot be formed using the type formers introduced so far, we must introduce new resources: the type former, token constructor and elimination rule for **identity types**.

As in the example above, the things we want to identity are tokens of some type. In set-theoretic mathematics we might ask of any two arbitrary entities whether they're identical, but in a type-theoretic framework this doesn't make sense: if two tokens aren't even of the same type then they certainly can't be identical. But for any two tokens $\mathtt{a} : \mathtt{C}$ and $\mathtt{b} : \mathtt{C}$ we can state the proposition that $\mathtt{a}$ and $\mathtt{b}$ are identical (which may be true or false). Thus the type former for the identity type must take as inputs a type $\mathtt{C}$ and two tokens $\mathtt{a} : \mathtt{C}$ and $\mathtt{b} : \mathtt{C}$. We write the type corresponding to the proposition that $\mathtt{a} : \mathtt{C}$ and $\mathtt{b} : \mathtt{C}$ are identical as:

$$\mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{b})$$

or sometimes as $\mathtt{a} =_{\mathtt{C}} \mathtt{b}$. As with product types in Section 4.1, the order of $\mathtt{a}$ and $\mathtt{b}$ matters, and $\mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{b})$ and $\mathtt{Id}_{\mathtt{C}}(\mathtt{b}, \mathtt{a})$ are different types.[59]

A token of $\mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{b})$ is a certificate to the proposition that the tokens $\mathtt{a}$ and $\mathtt{b}$ are identical. We call such a token an **identification** of $\mathtt{a}$ and $\mathtt{b}$. So, for example, (once we've defined the natural numbers $\mathbb{N}$) we can form the type $\mathtt{Id}_{\mathbb{N}}(5, 5)$, and moreover we are able to construct a token of this type since the corresponding proposition is true. We can also form the type $\mathtt{Id}_{\mathbb{N}}(5, 7)$, that says falsely that 5 is identical to 7, but of course we cannot form a token of this latter type. As ever, we cannot directly assert in the language of HoTT that an identification does not exist, but we can say that the existence of some token leads to contradiction. Thus we can define the type $\neg\mathtt{Id}_{\mathbb{N}}(5, 7)$, which says that natural numbers 5 and 7 are **distinct**, and moreover we can prove this by constructing a token of that type.

In the case of the natural numbers we can prove that, for any $\mathtt{m}, \mathtt{n} : \mathbb{N}$ we have

$$\mathtt{Id}_{\mathbb{N}}(\mathtt{m}, \mathtt{n}) + \neg\mathtt{Id}_{\mathbb{N}}(\mathtt{m}, \mathtt{n})$$

duction', from two other principles known in the literature as 'contractability' (which we call 'the uniqueness principle for identity types) and 'transport' (which we call 'substitution salva veritate'). The mathematical relationship we exploit is known, however, it is not standardly used to provide a justification and motivation for path induction in the way explained below. We think our approach is pedagogically and philosophically preferable for reasons discussed in our papers 'Identity in HoTT, Part I: the justification of path induction' and 'Does HoTT provide a foundation for mathematics?' available in draft form on the project website `http://www.bristol.ac.uk/arts/research/current-projects/homotopy-type-theory/`.

[59] In Section 7.5 we show that identity is commutative, i.e. that given a token of $\mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{b})$ we can produce a token of $\mathtt{Id}_{\mathtt{C}}(\mathtt{b}, \mathtt{a})$, and vice versa.

(i.e. any two natural numbers are either identical or distinct). However, we do not assume this for arbitrary types, and for some types $\mathtt{C}$ we may have tokens $\mathtt{a}$ and $\mathtt{b}$ for which we can neither produce a token of $\mathtt{Id_C(a,b)}$ nor a token of $\neg\mathtt{Id_C(a,b)}$.

Recall from Section 1.6 that under the Propositions as Types view propositions are not merely true or false, and inhabited types can have multiple certificates. This principle holds for identity types as well, and so we do not restrict identity types to being simply either inhabited or uninhabited. If $\mathtt{Id_C(a,b)}$ is inhabited then it may have multiple tokens: for a given pair of tokens $\mathtt{a}$ and $\mathtt{b}$ there may be multiple identifications of them.

7.1.1 Higher identity types

As an aside, and as a hint of things to come, we briefly consider an issue that will later be of central importance.

For every pair of tokens of any given type there is a type former for the corresponding identity type. In particular, then, there is such a type former for any pair of identifications in any identity type. That is, given $\mathtt{a : C}$ and $\mathtt{b : C}$, any two identifications $\mathtt{p, q : Id_C(a,b)}$ may themselves be identical or distinct. We can form the **higher identity type**

$$\mathtt{Id_{Id_C(a,b)}(p,q)}$$

whose tokens (if there are any) are identifications between $\mathtt{p}$ and $\mathtt{q}$. Similarly, we can iterate this procedure: given any two tokens $\mathtt{r}$ and $\mathtt{s}$ of $\mathtt{Id_{Id_C(a,b)}(p,q)}$ we can form the higher identity type

$$\mathtt{Id_{Id_{Id_C(a,b)}(p,q)}(r,s)}$$

whose tokens (if there are any) are identifications between $\mathtt{r}$ and $\mathtt{s}$; and so on.

In principle this iteration into higher and higher identity types may go on endlessly. It may be tempting, then, to introduce some new principle to artificially truncate this iteration – for example, by imposing an axiom that says that all higher identity types are 'trivial' in some sense. However, part of the founding insight of Homotopy Type Theory is that the structure of these higher identity types is deeply related to the study of spaces in *homotopy theory*, and it is from this discovery that HoTT derives much of its power and interest.[60]

However, none of these complications are required in order to understand the basics of the theory. In practice we will be concerned only with the existence or non-

[60] This insight is originally due to Awodey, Warren, and Voevodsky. For technical details see S. Awodey and M. A. Warren (2009) "Homotopy theoretic models of identity types", *Mathematical Proceedings of the Cambridge Philosophical Society*, 146, pp. 45–55.

existence of identifications between the mathematical entities that are of primary interest, and questions of whether these identifications are themselves identical or distinct will not be important. We can therefore safely defer consideration of higher identity types (and other constructions involving more advanced uses of identity types) to part V of the present work.

7.2 Token constructor for the identity type

The token constructor for the identity type provides the identifications that must be included as part of the definition of identity. It's clear that we shouldn't just be able to freely create tokens of $\mathtt{Id_C(a,b)}$ for arbitrary $\mathtt{a}$ and $\mathtt{b}$, since this corresponds to proving that any two tokens of a type are identical (for example, that two arbitrary natural numbers are identical). The only identifications that are guaranteed to exist (with no further assumptions or premises) are the *trivial self-identities*. So for any type $\mathtt{C}$ we must have a certificate to the proposition that each token of $\mathtt{C}$ is self identical – i.e. that identity is **reflexive**. The certificate for the reflexivity of identity is a (polymorphic) dependent function

$$\mathtt{refl} : \prod_{\mathtt{C:TYPE}} \prod_{\mathtt{x:C}} \mathtt{Id_C(x,x)}$$

which takes a type $\mathtt{C}$ and a token $\mathtt{x : C}$ as inputs and returns as output a token $\mathtt{refl_x : Id_C(x,x)}$.[61] This is the token constructor for the identity type, and it allows us to construct one certificate to $\mathtt{Id_C(x,x)}$ for each $\mathtt{x : C}$, and nothing else.

Since we can only produce trivial self-identifications, the identity type may appear to be of little use: what good is an identity sign if we can only assert things of the form $a = a$ and $x = x$, etc.? However, the situation is more subtle than this, as we show in the next section.

Before we get to that, note that there's a way we can use identity types even without being able to construct tokens of them, since we can *posit* identifications as premises or assumptions. For example, in a calculation we might want to set two variables $\mathtt{j : C}$ and $\mathtt{k : C}$ to have identical values by positing that the corresponding type is inhabited, i.e. by taking a token of $\mathtt{Id_C(j,k)}$ as a premise. We can then use this posit to derive further consequences (which may include the existence of other identifications of other tokens). We thereby prove that if $\mathtt{j} =_{\mathtt{C}} \mathtt{k}$ then the derived consequences follow, perhaps without being able to prove that $\mathtt{j}$ and $\mathtt{k}$ are in fact identical.

[61] For brevity, and since no ambiguity can arise, we omit the type from the subscript to $\mathtt{refl}$, writing $\mathtt{refl_x}$ rather than $\mathtt{refl_{C,x}}$.

However, it still appears that the only identifications that we can get hold of are the trivial self-identifications, since these are the only ones given to us by the token constructors. To see why this is not the case we must re-examine the relationship between the token constructors for a type and the possible range of tokens of that type.

7.3 Uniqueness principles

When we previously introduced new way of forming types, such as coproduct types or the dependent function types, we (implicitly) assumed that the relevant token constructors give us all the tokens of those types. For example, we assumed that every token in $\mathtt{A} + \mathtt{B}$ is either $\mathtt{inl(a)}$ for some $\mathtt{a} : \mathtt{A}$ or $\mathtt{inr(b)}$ for some $\mathtt{b} : \mathtt{B}$, because $\mathtt{inl}$ and $\mathtt{inr}$ are the two constructors for the coproduct. Likewise, we have assumed that every token of $\mathtt{A} \times \mathtt{B}$ is of the form $(\mathtt{a}, \mathtt{b})$ for some $\mathtt{a} : \mathtt{A}$ and some $\mathtt{b} : \mathtt{B}$.

However, now that we have introduced identity types we can refine this assumption and make it more precise: we can say formally within the language of HoTT that every token of the types introduced so far is *identical* to the output of a corresponding token constructor. Thus for every token $\mathtt{c} : \mathtt{A} + \mathtt{B}$ we have either a token of type $\mathtt{Id}_{\mathtt{A}+\mathtt{B}}(\mathtt{c}, \mathtt{inl(a)})$ for some $\mathtt{a} : \mathtt{A}$ or a token of type $\mathtt{Id}_{\mathtt{A}+\mathtt{B}}(\mathtt{c}, \mathtt{inr(b)})$ for some $\mathtt{b} : \mathtt{B}$.[62] We can therefore express these statements formally within the language of HoTT, and for the types defined in Sections 4 and 6 we can prove them.

So the correct interpretation of the role of the token constructors introduced so far is not that they give us literally every token of the relevant type, but rather that they give us every token of the relevant type *up to identity*. We should not say that every token of the types introduced so far is the output of one of the constructors, but rather that every such token is *identical to* the output of one of the constructors, where the identiity is witnessed by a token of the appropriate identity type. This is an important distinction. For example, it means that we cannot guarantee even that the Unit type has just a single token, namely $* : \mathtt{1}$. Rather, we can only say that if there are any further tokens of the Unit type then they must be identical to $*$, i.e.

$$\prod_{\mathtt{x}:\mathtt{1}} \mathtt{Id}_{\mathtt{1}}(\mathtt{x}, *)$$

must be inhabited. There may in fact be a very large number of tokens of $\mathtt{1}$, but since the above type is inhabited they are all identical as tokens of $\mathtt{1}$.

[62] Exercise: write out this statement formally using Π- and Σ-types.

As we explain below, this means that there is nothing we can say in the language of HoTT to distinguish such tokens. Thus, relative to the language of HoTT itself, there is simply no fact of the matter about how many tokens any particular type has, there are only facts about the number of tokens *up to identity*.

7.4 Uniqueness principle for Identity types

What is the corresponding uniqueness principle for the identity type itself? Since the only token constructor we have is `refl` it seemed that the only tokens of identity types that we could have were the trivial self-identifications of the form $\mathtt{refl}_{\mathtt{x}}$ for some $\mathtt{x} : \mathtt{C}$. But now, in light of the above discussion, we should modify this statement. The constructor `refl` doesn't give us all the identifications, but rather all the identifications *up to identity* in the appropriate type.

However, we need to take care here, because the obvious uniqueness principle – that every identification is identical to the output of the token constructor – cannot be correct. We can't prove that every arbitrary identification $\mathtt{p} : \mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{b})$ is identical to $\mathtt{refl}_{\mathtt{x}} : \mathtt{Id}_{\mathtt{C}}(\mathtt{x}, \mathtt{x})$ for some $\mathtt{x}$, because only tokens that live in the same type can be identified. To state an appropriate uniqueness principle for identifications we must therefore introduce a type in which arbitrary identifications and trivial self-identities can be found together; or rather (since each token belongs to just one type) in which suitable *counterparts* of these identifications can be found together.

Recall from Section 6.5 that any relation taking two arguments can be partially applied to a single argument to give a predicate. Thus for any token $\mathtt{a} : \mathtt{C}$ we can define the predicate

$$\mathtt{Id}_{\mathtt{C}}(\mathtt{a}) : \mathtt{C} \to \mathtt{TYPE}$$

which takes a token $\mathtt{b} : \mathtt{C}$ as input and returns the type $\mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{b})$ as output. This is therefore the predicate that asserts of any token in $\mathtt{C}$ that it is identical to $\mathtt{a}$.

With this predicate we can form a dependent pair type

$$\mathtt{E}_{\mathtt{a}} :\equiv \sum_{\mathtt{x}:\mathtt{C}} \mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{x})$$

Read as a proposition, this says that *some* token $\mathtt{x} : \mathtt{C}$ is identical to $\mathtt{a}$. This is of course trivially true, and we always have the token $(\mathtt{a}, \mathtt{refl}_{\mathtt{a}})$ as a certificate to it. More generally, the tokens of $\mathtt{E}_{\mathtt{a}}$ are pairs $(\mathtt{b}, \mathtt{p})$, where $\mathtt{b} : \mathtt{C}$ and $\mathtt{p} : \mathtt{Id}_{\mathtt{C}}(\mathtt{a}, \mathtt{b})$. This therefore gives us a type in which we can find (counterparts to) both arbitrary identifications and a trivial self-identification. Moreover, for any identification $\mathtt{q} : \mathtt{Id}_{\mathtt{C}}(\mathtt{s}, \mathtt{t})$ there is a dependent pair type $\mathtt{E}_{\mathtt{s}}$ in which (a counterpart to) that identification can be found. These dependent pair types therefore make it possible to state a suitable uniqueness principle for identity types.

Recall from Section 6.4 that we can read Σ-types in two different ways. Aside from the reading of $\sum_{\mathtt{x}:\mathtt{C}} \mathtt{Id}_{\mathtt{C}}(\mathtt{a},\mathtt{x})$ as an existentially quantified proposition we can also read it as the subtype $\{\mathtt{x}:\mathtt{C} \,|\, \mathtt{Id}_{\mathtt{C}}(\mathtt{a},\mathtt{x})\}$, i.e. the type of *tokens of* `C` *that are identical to* `a`. In set theory this would just be the singleton of `a`.

The appropriate uniqueness principle for identity types therefore doesn't say that the only token of $\mathtt{E}_{\mathtt{a}}$ is $(\mathtt{a},\mathtt{refl}_{\mathtt{a}})$, but rather that this is the only token *up to identity* in $\mathtt{E}_{\mathtt{a}}$. That is, every token $(\mathtt{b},\mathtt{p}):\mathtt{E}_{\mathtt{a}}$ is identical to $(\mathtt{a},\mathtt{refl}_{\mathtt{a}})$ in $\mathtt{E}_{\mathtt{a}}$. More formally we can say that for every type `C`, for every $\mathtt{a}:\mathtt{C}$, and for every token $(\mathtt{b},\mathtt{p}):\mathtt{E}_{\mathtt{a}}$, the type

$$\mathtt{Id}_{\mathtt{E}_{\mathtt{a}}}\big((\mathtt{a},\mathtt{refl}_{\mathtt{a}}),(\mathtt{b},\mathtt{p})\big)$$

is inhabited. This is the uniqueness principle for identity types.

To summarise: since the only token constructor for the identity type is `refl` it seemed at first that we could only use identity types to talk about trivial self-identifications. But this was based on the assumption that the only identifications that we can get hold of are those given by the constructor `refl` itself. Then, with a more refined understanding of uniqueness principles, we saw that this assumption is not quite correct: the most we can say is that every identification is identical to something provided by a constructor, *up to identity* in an appropriate type. Finally, we found that the type $\mathtt{E}_{\mathtt{a}}$ provides an appropriate type in which (counterparts of) arbitrary identifications and self-identities can be found, and using this type an appropriate uniqueness principle for identity types can be stated.

7.5 Substitution of identicals for identicals

To complete the definition of identity types we must give the elimination rule. In this section we start from a simple intuitively-justified principle and show that a surprising elimination rule called *based path induction* follows from it, which is the elimination rule for identity types.

If a token of an identity type $\mathtt{Id}_{\mathtt{C}}(\mathtt{a},\mathtt{b})$ is supposed to be a certificate to the fact that `a` and `b` are identical then there are some basic requirements that identity types must satisfy. One of these is that if `s` and `t` are identical then anything that's true of one is true of the other. This is the principle of **substitution salva veritate** – we can substitute `s` for `t` in any statement while *saving the truth* of that statement.

We formalise this as follows: for any type `A`, any predicate $\mathtt{K}:\mathtt{A}\to\mathtt{TYPE}$ and any $\mathtt{s}:\mathtt{A}$ and $\mathtt{t}:\mathtt{A}$, we have a function of type

$$\mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{t})\times\mathtt{K}(\mathtt{t})\to\mathtt{K}(\mathtt{s})$$

That is, given an identification of $\mathtt{s}$ with $\mathtt{t}$ and a certificate to a proposition about $\mathtt{t}$, we can produce a certificate to the corresponding proposition about $\mathtt{s}$.

We can formalise the quantification over $\mathtt{A}$ by using Π-types: thus, for any type $\mathtt{A}$ and any predicate $\mathtt{K}$ on $\mathtt{A}$ we have a function

$$\mathtt{ssv}_{\mathtt{A},\mathtt{K}} : \prod_{\mathtt{s}:\mathtt{A}} \prod_{\mathtt{t}:\mathtt{A}} \mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{t}) \times \mathtt{K}(\mathtt{t}) \to \mathtt{K}(\mathtt{s})$$

Furthermore, for any $\mathtt{s} : \mathtt{A}$, when the function $\mathtt{ssv}_{\mathtt{A},\mathtt{K}}(\mathtt{s},\mathtt{s}) : \mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{s}) \times \mathtt{K}(\mathtt{s}) \to \mathtt{K}(\mathtt{s})$ is given $\mathtt{refl}_{\mathtt{s}} : \mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{s})$ as its first argument, we require that the resulting function of type $\mathtt{K}(\mathtt{s}) \to \mathtt{K}(\mathtt{s})$ is the identity function on $\mathtt{K}(\mathtt{s})$ that leaves its input unchanged. Note that we do *not* require that applying $\mathtt{ssv}_{\mathtt{A},\mathtt{K}}(\mathtt{s},\mathtt{s})$ to a token of $\mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{s})$ other than $\mathtt{refl}_{\mathtt{s}}$ gives the identity function on $\mathtt{K}(\mathtt{s})$. That is, the non-trivial identification of $\mathtt{s}$ with itself may lead to a corresponding non-trivial mapping of $\mathtt{K}(\mathtt{s})$ to itself. However, substituting $\mathtt{s}$ for itself and using the *trivial* self-identification of $\mathtt{s}$ as the certificate corresponds to making no substitution at all. Thus, whatever token of $\mathtt{K}(\mathtt{s})$ we had before this trivial substitution, we must have the same token of $\mathtt{K}(\mathtt{s})$ afterwards.

From the principle of substitution salva veritate we can derive other principles. If we let $\mathtt{K}$ be $\mathtt{Id}_{\mathtt{A}}(\mathtt{t})$ then we obtain

$$\mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{t}) \times \mathtt{Id}_{\mathtt{A}}(\mathtt{t},\mathtt{t}) \to \mathtt{Id}_{\mathtt{A}}(\mathtt{t},\mathtt{s})$$

and since we always have a token of $\mathtt{Id}_{\mathtt{A}}(\mathtt{t},\mathtt{t})$ this is just a statement of the symmetry of identity:

$$\mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{t}) \to \mathtt{Id}_{\mathtt{A}}(\mathtt{t},\mathtt{s})$$

Similarly, for any $\mathtt{u} : \mathtt{A}$, if we let the predicate $\mathtt{K}$ be $\mathtt{Id}_{\mathtt{A}}(\mathtt{u})$ then we obtain

$$\mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{t}) \times \mathtt{Id}_{\mathtt{A}}(\mathtt{u},\mathtt{t}) \to \mathtt{Id}_{\mathtt{A}}(\mathtt{u},\mathtt{s})$$

which, after applying symmetry, expresses the transitivity of identity:

$$\mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{t}) \times \mathtt{Id}_{\mathtt{A}}(\mathtt{t},\mathtt{u}) \to \mathtt{Id}_{\mathtt{A}}(\mathtt{s},\mathtt{u})$$

These are both straightforward and expected properties of identity. However, we can also derive an unexpected property of identity, as we show in the next section.

7.6 Based path induction

For some type $\mathtt{C}$ and some $\mathtt{a} : \mathtt{C}$, consider an arbitrary predicate $\mathtt{K} : \mathtt{E}_{\mathtt{a}} \to \mathtt{TYPE}$, where $\mathtt{E}_{\mathtt{a}} :\equiv \sum_{\mathtt{x}:\mathtt{C}} \mathtt{Id}_{\mathtt{C}}(\mathtt{a},\mathtt{x})$ as above. The principle of *substitution salva veritate* says

$$\mathtt{Id}_{\mathtt{E}_{\mathtt{a}}}(\mathtt{s},\mathtt{t}) \times \mathtt{K}(\mathtt{t}) \to \mathtt{K}(\mathtt{s})$$

for any tokens $\mathtt{s}, \mathtt{t} : \mathtt{E_a}$. Thus, in particular, if we take $(\mathtt{a}, \mathtt{refl_a}) : \mathtt{E_a}$ for $\mathtt{t}$ and an arbitrary token $(\mathtt{b}, \mathtt{p}) : \mathtt{E_a}$ for $\mathtt{s}$ we get

$$\mathtt{Id_{E_a}}\big((\mathtt{b},\mathtt{p}),\, (\mathtt{a},\mathtt{refl_a})\big) \times \mathtt{K}(\mathtt{a},\mathtt{refl_a}) \to \mathtt{K}(\mathtt{b},\mathtt{p})$$

Recall from the previous section that the uniqueness principle for identity types says that for every token $(\mathtt{b}, \mathtt{p}) : \mathtt{E_a}$ the type $\mathtt{Id_{E_a}}\big((\mathtt{a}, \mathtt{refl_a}), (\mathtt{b}, \mathtt{p})\big)$ is inhabited, and thus by symmetry of identity so is $\mathtt{Id_{E_a}}\big((\mathtt{b}, \mathtt{p}), (\mathtt{a}, \mathtt{refl_a})\big)$. So for any predicate $\mathtt{K} : \mathtt{E_a} \to \mathtt{TYPE}$ and any token $(\mathtt{b}, \mathtt{p}) : \mathtt{E_a}$ we have a function

$$\mathtt{K}(\mathtt{a}, \mathtt{refl_a}) \to \mathtt{K}(\mathtt{b}, \mathtt{p})$$

This says that to prove that some property holds for any identification $\mathtt{p} : \mathtt{Id_A}(\mathtt{a}, \mathtt{b})$ between $\mathtt{a} : \mathtt{A}$ and any token $\mathtt{b} : \mathtt{A}$, it suffices to prove that the property holds for $\mathtt{refl_a}$. This principle, called **based path induction**, is the elimination rule for the identity type.[63]

We can express this formally by replacing the informal quantifications with $\prod$-types quantifying over the corresponding types. The fully formal statement of based path induction is therefore:

$$\mathtt{BPI} : \prod_{\mathtt{A:TYPE}} \prod_{\mathtt{a:A}} \prod_{\mathtt{K:E_a \to TYPE}} \left[\mathtt{K}(\mathtt{a}, \mathtt{refl_a}) \to \prod_{(\mathtt{b},\mathtt{p}):\mathtt{E_a}} \mathtt{K}(\mathtt{b}, \mathtt{p}) \right]$$

This says that for any type $\mathtt{A}$, any $\mathtt{a} : \mathtt{A}$, any predicate $\mathtt{K} : \mathtt{E_a} \to \mathtt{TYPE}$ and any $\mathtt{y} : \mathtt{K}(\mathtt{a}, \mathtt{refl_a})$ we have a dependent function

$$\mathtt{BPI}(\mathtt{A}, \mathtt{a}, \mathtt{K}, \mathtt{y}) : \prod_{(\mathtt{b},\mathtt{p}):\mathtt{E_a}} \mathtt{K}(\mathtt{b}, \mathtt{p})$$

To complete the formal statement of based path induction we must state one extra feature: when the above function is applied to $(\mathtt{a}, \mathtt{refl_a})$ itself, it should return just the witness $\mathtt{y} : \mathtt{K}(\mathtt{a}, \mathtt{refl_a})$ that it was given. That is,

$$\mathtt{BPI}(\mathtt{A}, \mathtt{a}, \mathtt{K}, \mathtt{y})(\mathtt{a}, \mathtt{refl_a}) :\equiv \mathtt{y}$$

This is the *uniformity property* of based path induction.

The formal statement of based path induction is complicated and cumbersome. Happily for practical purposes we will almost never have to use it directly. Path induction allows us to simplify a proof that some property holds of all identifications in a particular identity type, just as mathematical induction allows us to

[63] The reason for calling it based *path* induction will be explained in Part IV when we introduce the homotopy interpretation for the formal type theory.

simplify a proof that some property holds of all natural numbers. In an induction on the natural numbers we first prove that the property holds of 0 (the base case), then show that if it holds of some n then it also holds of $n+1$ (the inductive step). A proof using path induction is even simpler than this: here only a base case needs to be proved, since `BPI` says that the inductive step holds automatically for any property that we might consider. Based path induction is therefore a powerful rule that makes many proofs involving identifications quite trivial. But, as we have seen, the rule itself follows directly from the principle of *substitution salva veritate* and the uniqueness principle for identity types.