Principles of Dependent Type Theory

Carlo Angiuli
Indiana University
cangiuli@indiana.edu

Daniel Gratzer
Aarhus University
gratzer@cs.au.dk

(2024-02-28)

My dear, here we must run as fast
as we can, just to stay in place.
And if you wish to go anywhere
you must run twice as fast as that.

Lewis Carroll, *Alice in Wonderland*

Acknowledgements

We thank Lars Birkedal for his comments and suggestions on drafts of these notes. We also thank the students who participated in *Modern Dependent Types* (CSCI-B619) at Indiana University and *Modern Dependent Type Theory* at Aarhus University in Spring 2024, for whom these notes were prepared. A special thanks to Mathias Adams Møller, Yafei Yang

add names of students here as people point out typos

Contents

1 Introduction

In these lecture notes, we aim to introduce the reader to a modern research perspective on the design of "full-spectrum" dependent type theories. At the end of this course, readers should be prepared to engage with contemporary research papers on dependent type theory, and to understand the motivations behind recent extensions of Martin-Löf's dependent type theory [ML84], including observational type theory [AMS07], homotopy type theory [UF13], and cubical type theory [CCHM18; Ang+21].

These lecture notes are in an early draft form and are missing many relevant citations. The authors welcome any feedback.

Dependent type theory (henceforth just *type theory*) often appears arcane to outside observers for a handful of reasons. First, as in the parable of the elephant, there are myriad perspectives on type theory. The language presented in these lecture notes, *mutatis mutandis*, can be accurately described as:

- the core language of assertions and proofs in *proof assistants* like Agda [Agda], Coq [Coq], Lean [MU21], and Nuprl [Con+85];
- a richly-typed *functional programming language*, as in Idris [Bra13] and Pie [FC18], as well as in the aforementioned proof assistants Agda and Lean [Chr23].
- an *axiom system* for reasoning synthetically in a number of mathematical settings, including locally cartesian closed 1-categories [Hof95], homotopy types [Shu21], and Grothendieck ∞-topoi [Shu19];
- a structural [Tse17], constructive [ML82] *foundation for mathematics* as an alternative to ZFC set theory [Alt23].

A second difficulty is that it is quite complex to even *define* type theory in a precise fashion, for reasons we shall discuss in Section 2.2, and the relative merits of different styles of definition—and even which ones satisfactorily define any object whatsoever—have been the subject of great debate among experts over the years.

Finally, much of the literature on type theory is highly technical—involving either lengthy proofs by induction or advanced mathematical machinery—in order to account for its complex definition and applications. In these lecture notes we attempt to split the difference by presenting a mathematically-informed viewpoint on type theory while avoiding advanced mathematical prerequisites.

Goals of the course As researchers who work on designing new type theories, our goal in this course is to pose and begin to answer the following questions: *What makes a good type theory, and why are there so many?* We will focus on *notions of equality in Martin-Löf type theory* as a microcosm of this broader question, studying how extensional [ML82], intensional [ML75], observational [AMS07; SAG22; PT22], homotopy [UF13], and cubical type theories [CCHM18; Ang+21] have provided increasingly sophisticated answers to this deceptively simple question.

In this chapter In Section 1.1 we introduce and motivate the concepts of type and term dependency, definitional equality, and propositional equality through the lens of typed functional programming. Note that Chapter 2 is self-contained albeit lacking in motivation, so readers unfamiliar with functional programming can safely skip ahead.

Goals of the chapter By the end of this chapter, you will be able to:

- Give examples of full-spectrum dependency.
- Explain the role of definitional equality in type-checking, and how and why it differs from ordinary closed-term evaluation.
- Explain the role of propositional equality in type-checking.

1.1 *Dependent types for functional programmers*

The reader is forewarned that the following section assumes some familiarity with functional programming, unlike the remainder of the lecture notes.

Types in programming languages For the purposes of this course, one should regard a programming language's (static) type system as its *grammar*, not as one of many potential static analyses that might be enabled or disabled.[1] Indeed, just as a parser may reject as nonsense a program whose parentheses are mismatched, or an untyped language's interpreter may reject as nonsense a program containing unbound identifiers, a type-checker may reject as nonsense the program `1 + "hi"` on the grounds that—much like the previous two examples—there is no way to successfully evaluate it.

Concretely, a type system divides a language's well-parenthesized, well-scoped expressions into a collection of sets: the *expressions of type* **Nat** are those that "clearly" compute natural numbers, such as literal natural numbers (0, 1, 120), arithmetic expressions (1 + 1),

[1] The latter perspective is valid, but we wish to draw a sharp distinction between types *qua* (structural) grammar, and static analyses that may be non-local, non-structural, or non-substitutive in nature.

and fully-applied functions that return natural numbers (fact `5`, atoi `"120"`). Similarly, the expressions of type **String** are those that clearly compute strings (`"hi"`, itoa `5`), and for any types A and B, the expressions of type $A \to B$ are those that clearly compute functions that, when passed an input of type A, clearly compute an output of type B.

What do we mean by "clearly"? One typically insists that type-checking be fully automated, much like parsing and identifier resolution. Given that determining the result of a program is in general undecidable, any automated type-checking process will necessarily compute a conservative underapproximation of the set of programs that compute (e.g.) natural numbers. (Likewise, languages may complain about unbound identifiers even in programs that can be evaluated without a runtime error!)

The goal of a type system is thus to rule out as many undesirable programs as possible without ruling out too many desirable ones, where both of these notions are subjective depending on which runtime errors one wants to rule out and which programming idioms one wants to support. Language designers engage in the neverending process of refining their type systems to rule out more errors and accept more correct code; full-spectrum dependent types can be seen as an extreme point in this design space.

1.1.1 *Uniform dependency: length-indexed vectors*

Every introduction to dependent types starts with the example of vectors, or lists with specified length. We start one step earlier by considering lists with a specified type of elements, a type which already exhibits a basic form of dependency.

Parameterizing by types One of the most basic data structures in functional programming languages is the *list*, which is either empty (written []) or consists of an element x adjoined to a list xs (written $x :: xs$). In typed languages, we typically require that a list's elements all have the same type so that we know what operations they support.

The simplest way to record this information is to have a separate type of lists for each type of element: a **ListOfNats** is either empty or a **Nat** and a **ListOfNats**, a **ListOfStrings** is either empty or a **String** and a **ListOfStrings**, etc. This strategy clearly results in repetition at the level of the type system, but it also causes code duplication because operations that work uniformly for any type of elements (e.g., reversing a list) must be defined twice for the two apparently unrelated types **ListOfNats** and **ListOfStrings**.

In much the same way that functions—terms indexed by terms—promote code reuse by allowing programmers to write a series of operations once and perform them on many different inputs, we can solve both problems described above by allowing types and terms to be uniformly parameterized by types. Thus the types **ListOfNats** and **ListOfStrings** become two instances (**List Nat** and **List String**) of a single family of types **List**:[2]

[2]For the time being, the reader should understand A : **Set** as notation meaning "A is a type."

$$
\begin{array}{l}
\textbf{data List}\ (A : \textbf{Set}) : \textbf{Set where} \\
\quad [\,] : \textbf{List}\ A \\
\quad \_{::}\_ : A \to \textbf{List}\ A \to \textbf{List}\ A
\end{array}
$$

and any operation that works for all element types A, such as returning the first (or all but first) element of a list, can be written as a family of operations:

$$
\begin{array}{l}
\mathsf{head} : (A : \textbf{Set}) \to \textbf{List}\ A \to A \\
\mathsf{head}\ A\ [\,] = \mathsf{error}\ \texttt{"List must be non-empty."} \\
\mathsf{head}\ A\ (x :: xs) = x \\
\\
\mathsf{tail} : (A : \textbf{Set}) \to \textbf{List}\ A \to \textbf{List}\ A \\
\mathsf{tail}\ A\ [\,] = \mathsf{error}\ \texttt{"List must be non-empty."} \\
\mathsf{tail}\ A\ (x :: xs) = xs
\end{array}
$$

By partially applying head to its type argument, we see that head **Nat** has type **List Nat** $\to$ **Nat** and head **String** has type **List String** $\to$ **String**, and the expression $1 + (\mathsf{head}\ \textbf{Nat}\ (1 :: [\,]))$ has type **Nat** whereas $1 + (\mathsf{head}\ \textbf{String}\ (\texttt{"hi"} :: [\,]))$ is ill-typed because the second input to + has type **String**.

Parameterizing types by terms The perfectionist reader may find the **List** A type unsatisfactory because it does not prevent runtime errors caused by applying head and tail to the empty list []. We cannot simply augment our types to track which lists are empty, because $2 :: 1 :: [\,]$ and $1 :: [\,]$ are both nonempty but we can apply tail **Nat** twice to the former before encountering an error, but only once to the latter.

Instead, we parameterize the type of lists not only by their type of elements as before but also by their length—a *term* of type **Nat**—producing the following family of types:[3]

$$
\begin{array}{l}
\textbf{data Vec}\ (A : \textbf{Set}) : \textbf{Nat} \to \textbf{Set where} \\
\quad [\,] : \textbf{Vec}\ A\ 0 \\
\quad \_{::}\_ : \{n : \textbf{Nat}\} \to A \to \textbf{Vec}\ A\ n \to \textbf{Vec}\ A\ (\mathsf{suc}\ n)
\end{array}
$$

Types parameterized by terms are known as *dependent types*.

Now the types of concrete lists are more informative—$(2 :: 1 :: [\,]) : \textbf{Vec Int}\ 2$ and $(1 :: [\,]) : \textbf{Vec Int}\ 1$—but more importantly, we can give head and tail more informative types which rule out the runtime error of applying them to empty lists. We do so by revising their input type to **Vec** A (suc n) for some n : **Nat**, which is to say that the vector has length at least one, hence is nonempty:

[3]Curly braces $\{n : \textbf{Nat}\}$ indicate *implicit* arguments automatically inferred by the type-checker.

$$\text{head} : \{A : \textbf{Set}\}\ \{n : \textbf{Nat}\} \to \textbf{Vec}\ A\ (\text{suc}\ n) \to A$$

-- head [] is impossible

$$\text{head}\ (x :: xs) = x$$

$$\text{tail} : \{A : \textbf{Set}\}\ \{n : \textbf{Nat}\} \to \textbf{Vec}\ A\ (\text{suc}\ n) \to \textbf{Vec}\ A\ n$$

-- tail [] is impossible

$$\text{tail}\ (x :: xs) = xs$$

Consider now the operation that concatenates two vectors:

$$\text{append} : \{A : \textbf{Set}\}\ \{n : \textbf{Nat}\}\ \{m : \textbf{Nat}\} \to \textbf{Vec}\ A\ n \to \textbf{Vec}\ A\ m \to \textbf{Vec}\ A\ (n + m)$$

Unlike our previous examples, the output type of this function is indexed not by a variable A or n, nor a constant **Nat** or 0, nor even a constructor suc $-$, but by an *expression* $n + m$. This introduces a further complication, namely that we would like this expression to be simplified as soon as n and m are known. For example, if we apply append to two vectors of length one ($n = m = 1$), then the result will be a vector of length two ($n + m = 1 + 1 = 2$), and we would like the type system to be aware of this fact in the sense of accepting as well-typed the expression head (tail (append l l')) for l and l' of type **Vec Nat** 1.

Because head (tail x) is only well-typed when x has type **Vec** A (suc (suc n)) for some n : **Nat**, this condition amounts to requiring that the expression append l l' not only has type **Vec** A ((suc 0) + (suc 0)) as implied by the type of append, but also type **Vec** A (suc (suc 0)) as implied by its runtime behavior. In short, we would like the two type expressions **Vec** A $(1+1)$ and **Vec** A 2 to *denote the same type* by virtue of the fact that $1 + 1$ and 2 *denote the same value*. In practice, we achieve this by allowing the type-checker to *evaluate expressions in types during type-checking*.

In fact, the length of a vector can be any expression whatsoever of type **Nat**. Consider filter, which takes a function $A \to$ **Bool** and a list and returns the sublist for which the function returns true. If the input list has length n, what is the length of the output?

$$\text{filter} : \{A : \textbf{Set}\}\ \{n : \textbf{Nat}\} \to (A \to \textbf{Bool}) \to \textbf{Vec}\ A\ n \to \textbf{Vec}\ A\ \boxed{?}$$

After a moment's thought we realize the length is not a function of n at all, but rather a recursive function of the input function and list:

$$\text{filter} : \{A : \textbf{Set}\}\ \{n : \textbf{Nat}\} \to (f : A \to \textbf{Bool}) \to (l : \textbf{Vec}\ A\ n) \to \textbf{Vec}\ A\ (\text{filterLen}\ f\ l)$$

$$\text{filterLen} : \{A : \textbf{Set}\}\ \{n : \textbf{Nat}\} \to (A \to \textbf{Bool}) \to \textbf{Vec}\ A\ n \to \textbf{Nat}$$

$$\text{filterLen}\ f\ [\,] = 0$$

$$\text{filterLen}\ f\ (x :: xs) = \textbf{if}\ f(x)\ \textbf{then}\ \text{suc}\ (\text{filterLen}\ f\ xs)\ \textbf{else}\ \text{filterLen}\ f\ xs$$

As before, once f and l are known the type of $\mathsf{filter}\ f\ l : \mathbf{Vec}\ A\ (\mathsf{filterLen}\ f\ l)$ will simplify by evaluating $\mathsf{filterLen}\ f\ l$, but as long as either remains a variable we cannot learn much by computation. Nevertheless, filterLen has many properties of interest: $\mathsf{filterLen}\ f\ l$ is at most the length of l, $\mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ l$ is always 0 regardless of l, etc. We will revisit this point in Section 1.1.3.

Remark 1.1.1. If we regard **Nat** and + as a user-defined data type and recursive function, as type theorists are wont to do, then filter's type using filterLen is entirely analogous to append's type using +. We wish to emphasize that, whereas one could easily imagine arithmetic being a privileged component of the type system, filter demonstrates that type indices may need to contain arbitrary user-defined recursive functions. ⋄

Another approach? If our only goal was to eliminate runtime errors from head and tail, we might reasonably feel that dependent types have overcomplicated the situation—we needed to introduce a new function just to write the type of filter! And indeed there are simpler ways of keeping track of the length of lists, which we describe briefly here.

First let us observe that a lower bound on a list's length is sufficient to guarantee it is nonempty and thus that an application of head or tail will succeed; this allows us to trade precision for simplicity by restricting type indices to be arithmetic expressions. Secondly, in the above examples we can perform type-checking and "length-checking" in two separate phases, where the first phase replaces every occurrence of $\mathbf{Vec}\ A\ n$ with $\mathbf{List}\ A$ before applying a standard non-dependent type-checking algorithm. This is possible because we can regard the dependency in $\mathbf{Vec}\ A\ n$ as expressing a computable *refinement*—or subset—of the non-dependent type of lists, namely $\{l : \mathbf{List}\ A \mid \mathsf{length}\ l = n\}$.

Combining these insights, we can by and large automate length-checking by recasting the type dependency of **Vec** in terms of arithmetic inequality constraints over an ML-style type system, and checking these constraints with SMT solvers and other external tools. At a very high level, this is the approach taken by systems such as Dependent ML [Xi07] and Liquid Haskell [Vaz+14]. Dependent ML, for instance, type-checks the usual definition of filter at the following type, without any auxiliary filterLen definition:

$$\mathsf{filter} : \mathbf{Vec}\ A\ m \to (\{n : \mathbf{Nat} \mid n \leq m\} \times \mathbf{Vec}\ A\ n)$$

Refinement type systems like these have proven very useful in practice and continue to be actively developed, but we will not discuss them any further for the simple reason that, although they are a good solution to head/tail and many other examples, they cannot handle full-spectrum dependency as discussed below.

1.1.2 *Non-uniform dependency: computing arities*

Thus far, all our examples of (type- or term-) parameterized types are *uniformly* parameterized, in the sense that the functions **List** : **Set** → **Set** and **Vec** A : **Nat** → **Set** do not inspect their arguments; in contrast, ordinary term-level functions out of **Nat** such as fact : **Nat** → **Nat** can and usually do perform case-splits on their inputs. In particular, we have not yet considered any families of types in which the head, or top-level, type constructor (→, **Vec**, **Nat**, etc.) differs between indices.

A type theory is said to have full-spectrum dependency if it permits the use of *non-uniformly term-indexed* families of types, such as the following **Nat**-indexed family:

```
nary : Set → Nat → Set
nary A 0 = A
nary A (suc n) = A → nary A n
```

Although **Vec Nat** and nary **Nat** are both functions **Nat** → **Set**, the latter's head type constructor varies between indices: nary **Nat** 0 = **Nat** but nary **Nat** 1 = **Nat** → **Nat**.

Using nary to compute the type of n-ary functions, we can now define not only varadic functions but even higher-order functions taking variadic functions as input, such as apply which applies an n-ary function to a vector of length n:

```
apply : {A : Set} {n : Nat} → nary A n → Vec A n → A
apply x [] = x
apply f (x :: xs) = apply (f x) xs
```

For A = **Nat** and $n = 1$, apply applies a unary function **Nat** → **Nat** to the head element of a **Vec Nat** 1; for A = **Nat** and $n = 3$, it applies a ternary function **Nat** → **Nat** → **Nat** → **Nat** to the elements of a **Vec Nat** 3:

```
apply suc (1 :: []) : Nat   -- evaluates to 2
apply _+_ : Vec Nat 2 → Nat
apply _+_ (1 :: 2 :: []) : Nat   -- evaluates to 3
apply (λx y z → x + y + z) (1 :: 2 :: 3 :: []) : Nat   -- evaluates to 6
```

Although apply is not the first time we have seen a function whose type involves a different recursive function—we saw this already with filter—this is our first example of a function that cannot be straightforwardly typed in an ML-style type system. Another way to put it is that nary A n → **Vec** A n → A is not the refinement of an ML type because nary A n is sometimes but not always a function type.

Remark 1.1.2. For the sake of completeness, it is also possible to consider *non-uniformly type-indexed* families of types, which go by a variety of names including non-parametric polymorphism, intensional type analysis, and typecase [HM95]. These often serve as optimized implementations of uniformly type-indexed families of types; a classic non-type-theoretic example is the C++ family of types `std::vector` for dynamically-sized arrays, whose `std::vector<bool>` instance may be compactly implemented using bitfields. ⋄

To understand the practical ramifications of non-uniform dependency, we will turn our attention to a more complex example: a basic implementation of sprintf in Agda (Figure 1.1). This function takes as input a **String** containing format specifiers such as `%u` (indicating a **Nat**) or `%s` (indicating a **String**), as well as additional arguments of the appropriate type for each format specifier present, and returns a **String** in which each format specifier has been replaced by the corresponding argument rendered as a **String**.

```
sprintf "%s %u" "hi" 2 : String   -- evaluates to "hi 2"
sprintf "%s" : String → String
sprintf "nat %u then int %d then char %c" : Nat → Int → Char → String
sprintf "%u" 5 : String   -- evaluates to "5"
sprintf "%u%% of %s%c" 3 "GD" 'P' : String   -- evaluates to "3% of GDP"
```

Our implementation uses various types and functions imported from Agda's standard library, notably toList : **String** → **List Char** which converts a string to a list of characters (length-one strings `'x'`). It consists of four main components:

- a data type Token which enumerates all relevant components of the input **String**, namely format specifiers (such as natTok : Token for `%u` and strTok : Token for `%s`) and literal characters (char `'x'` : Token);
- a function lex which tokenizes the input string, represented as a **List Char**, from left to right into a **List** Token for further processing;
- a function args which converts a **List** Token into a function type containing the additional arguments that sprintf must take; and
- the sprintf function itself.

Let us begin by convincing ourselves that our first example type-checks:

```
sprintf "%s %u" "hi" 2 : String   -- evaluates to "hi 2"
```

Because sprintf : (s : **String**) → printfType s, the partial application sprintf `"%s %u"` has type printfType `"%s %u"`. By evaluation, the type-checker can see printfType `"%s %u"` = args (strTok :: char `' '` :: natTok :: []) = **String** → **Nat** → **String**. Thus sprintf `"%s %u"` : **String** → **Nat** → **String**, and the remainder of the expression type-checks easily.

Now let us consider the definition of sprintf, which uses a helper function loop : (*toks* : **List** Token) → **String** → args *toks* whose first argument stores the Tokens yet to

```
data Token : Set where
  char : Char → Token
  intTok : Token
  natTok : Token
  chrTok : Token
  strTok : Token

lex : List Char → List Token
lex [] = []
lex ('%' :: '%' :: cs) = char '%' :: lex cs
lex ('%' :: 'd' :: cs) = intTok :: lex cs
lex ('%' :: 'u' :: cs) = natTok :: lex cs
lex ('%' :: 'c' :: cs) = chrTok :: lex cs
lex ('%' :: 's' :: cs) = strTok :: lex cs
lex (c :: cs) = char c :: lex cs

args : List Token → Set
args [] = String
args (char _ :: toks) = args toks
args (intTok :: toks) = Int → args toks
args (natTok :: toks) = Nat → args toks
args (chrTok :: toks) = Char → args toks
args (strTok :: toks) = String → args toks

printfType : String → Set
printfType s = args (lex (toList s))

sprintf : (s : String) → printfType s
sprintf s = loop (lex (toList s)) ""
  where
  loop : (toks : List Token) → String → args toks
  loop [] acc = acc
  loop (char c :: toks) acc = loop toks (acc ++ fromList (c :: []))
  loop (intTok :: toks) acc = λi → loop toks (acc ++ showInt i)
  loop (natTok :: toks) acc = λn → loop toks (acc ++ showNat n)
  loop (chrTok :: toks) acc = λc → loop toks (acc ++ fromList (c :: []))
  loop (strTok :: toks) acc = λs → loop toks (acc ++ s)
```

Figure 1.1: A basic Agda implementation of sprintf.

be processed, and whose second argument is the **String** accumulated from printing the already-processed Tokens. What is needed to type-check the definition of loop? We can examine a representative case in which the next Token is natTok:

$$\mathsf{loop}\ (\mathsf{natTok} :: \mathit{toks})\ \mathit{acc} = \lambda n \to \mathsf{loop}\ \mathit{toks}\ (\mathit{acc} +\!\!+\ \mathsf{showNat}\ n)$$

Note that $\mathit{toks} : \mathbf{List}\ \mathsf{Token}$ and $\mathit{acc} : \mathbf{String}$ are (pattern) variables, and the right-hand side ought to have type $\mathsf{args}\ (\mathsf{natTok} :: \mathit{toks})$. We can type-check the right-hand side—given that $\_{+\!\!+}\_ : \mathbf{String} \to \mathbf{String} \to \mathbf{String}$ is string concatenation and $\mathsf{showNat} : \mathbf{Nat} \to \mathbf{String}$ prints a natural number—and observe that it has type $\mathbf{Nat} \to \mathsf{args}\ \mathit{toks}$ by the type of loop.

Type-checking this clause thus requires us to reconcile the right-hand side's expected type $\mathsf{args}\ (\mathsf{natTok} :: \mathit{toks})$ with its actual type $\mathbf{Nat} \to \mathsf{args}\ \mathit{toks}$. Although these type expressions are quite dissimilar—one is a function type and the other is not—the definition of args contains a promising clause:

$$\mathsf{args}\ (\mathsf{natTok} :: \mathit{toks}) = \mathbf{Nat} \to \mathsf{args}\ \mathit{toks}$$

As in our earlier example of $\mathbf{Vec}\ A\ (1{+}1)$ and $\mathbf{Vec}\ A\ 2$ we would like the type expressions $\mathsf{args}\ (\mathsf{natTok} :: \mathit{toks})$ and $\mathbf{Nat} \to \mathsf{args}\ \mathit{toks}$ to denote the same type, but unlike the equation $1 + 1 = 2$, here both sides contain a free variable *toks* so we cannot appeal to evaluation, which is a relation on *closed* terms (ones with no free variables).

One can nevertheless imagine some form of *symbolic evaluation* relation that extends evaluation to open terms and *can* equate these two expressions. In this particular case, this step of closed evaluation is syntactically indifferent to the value of *toks* and thus can be safely applied even when *toks* is a variable. (Likewise, to revisit an earlier example, the equation $\mathsf{filterLen}\ f\ [\,] = 0$ should hold even for variable f.)

Thus we would like the type expressions $\mathsf{args}\ (\mathsf{natTok} :: \mathit{toks})$ and $\mathbf{Nat} \to \mathsf{args}\ \mathit{toks}$ to denote the same type by virtue of the fact that they *symbolically evaluate to the same symbolic value*, and to facilitate this we must allow the type-checker to *symbolically evaluate* expressions in types during type-checking. The congruence relation on expressions so induced is known as *definitional equality* because it contains defining clauses like this one.

Remark 1.1.3. Semantically we can justify this equation by observing that for any closed instantiation toks of *toks*, $\mathsf{args}\ (\mathsf{natTok} :: \mathrm{toks})$ and $\mathbf{Nat} \to \mathsf{args}\ \mathrm{toks}$ will evaluate to the same type expression—at least, once we have defined evaluation of type expressions—and thus this equation always holds at runtime. But just as (for reasons of decidability) the condition "when this expression is applied to a natural number it evaluates to a natural number" is a necessary but not sufficient condition for type-checking at $\mathbf{Nat} \to \mathbf{Nat}$, we do not want to take this semantic condition as the definition of definitional equality. It is however a necessary condition assuming that the type system is sound for the given evaluation semantics. (See Section 3.3.) ⋄

Definitional equality is the central concept in full-spectrum dependent type theory because it determines which types are equal and thus which terms have which types. In practice, it is typically defined as the congruence closure of the β-like reductions (also known as $\beta\delta\zeta\iota$-reductions) plus η-equivalence at some types; see Chapter 2 for details.

1.1.3 *Proving type equations*

Unfortunately, in light of Remark 1.1.3, there are many examples of type equations that are not direct consequences of ordinary or even symbolic evaluation. On occasion these equations are of such importance that researchers may attempt to make them definitional—that is, to include them in the definitional equality relation and adjust the type-checking algorithm accordingly [AMB13]. But such projects are often major research undertakings, and there are even examples of equations that can be definitional but are in practice best omitted due to efficiency or usability issues [Alt+01].

Let us turn once again to the example of filter from Section 1.1.2.

$$\mathsf{filter} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to (f : A \to \mathbf{Bool}) \to (l : \mathbf{Vec}\,A\,n) \to \mathbf{Vec}\,A\,(\mathsf{filterLen}\,f\,l)$$

$$\begin{aligned}
&\mathsf{filterLen} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to (A \to \mathbf{Bool}) \to \mathbf{Vec}\ A\ n \to \mathbf{Nat}\\
&\mathsf{filterLen}\ f\ [\,] = 0\\
&\mathsf{filterLen}\ f\ (x :: xs) = \mathbf{if}\ f(x)\ \mathbf{then}\ \mathsf{suc}\ (\mathsf{filterLen}\ f\ xs)\ \mathbf{else}\ \mathsf{filterLen}\ f\ xs
\end{aligned}$$

Suppose for the sake of argument that we want the operation of filtering an arbitrary vector by the constantly false predicate to return a $\mathbf{Vec}\ A\ 0$:

$$\begin{aligned}
&\mathsf{filterAll} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to \mathbf{Vec}\ A\ n \to \mathbf{Vec}\ A\ 0\\
&\mathsf{filterAll}\ l = \mathsf{filter}\ (\lambda x \to \mathsf{false})\ l \quad \text{-- does not type-check}
\end{aligned}$$

The right-hand side above has type $\mathbf{Vec}\,A\,(\mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ l)$ rather than $\mathbf{Vec}\,A\,0$ as desired, and in this case the expression $\mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ l$ cannot be simplified by (symbolic) evaluation because filterLen computes by recursion on l which is a variable. However, by induction on the possible instantiations of $l : \mathbf{Vec}\ A\ n$, either:

- $l = [\,]$, in which case $\mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ [\,]$ is definitionally equal (in fact, evaluates) to 0; or

- $l = x :: xs$, in which case we have the definitional equalities

$$\begin{aligned}
&\mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ (x :: xs)\\
=\ &\mathbf{if}\ \mathsf{false}\ \mathbf{then}\ \mathsf{suc}\ (\mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ xs)\ \mathbf{else}\ \mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ xs\\
=\ &\mathsf{filterLen}\ (\lambda x \to \mathsf{false})\ xs
\end{aligned}$$

for any x and xs. By the inductive hypothesis on xs, filterLen $(\lambda x \to \text{false})\ xs = 0$ and thus filterLen $(\lambda x \to \text{false})\ (x :: xs) = 0$ as well.

By adding a type of *provable equations* $a \equiv b$ to our language, we can compactly encode this inductive proof as a recursive function computing filterLen $(\lambda x \to \text{false})\ l \equiv 0$:

$$
\begin{aligned}
&\_\!\equiv\!\_ : \{A : \mathbf{Set}\} \to A \to A \to \mathbf{Set}\\
&\mathbf{refl} : \{A : \mathbf{Set}\}\ \{x : A\} \to x \equiv x\\
\\
&\mathsf{lemma} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to (l : \mathbf{Vec}\ A\ n) \to \mathsf{filterLen}\ (\lambda l \to \mathsf{false})\ l \equiv 0\\
&\mathsf{lemma}\ [\,] = \mathbf{refl}\\
&\mathsf{lemma}\ (x :: xs) = \mathsf{lemma}\ xs
\end{aligned}
$$

The [] clause of lemma ought to have type filterLen $(\lambda l \to \text{false})\ [\,] \equiv 0$, which is definitionally equal to the type $0 \equiv 0$ and thus **refl** type-checks. The $(x :: xs)$ clause must have type filterLen $(\lambda l \to \text{false})\ (x :: xs) \equiv 0$, which is definitionally equal to filterLen $(\lambda l \to \text{false})\ xs \equiv 0$, the expected type of the recursive call lemma xs.

Now armed with a function lemma that constructs for any $l : \mathbf{Vec}\ A\ n$ a proof that filterLen $(\lambda l \to \text{false})\ l \equiv 0$, we can justify *casting* from the type $\mathbf{Vec}\ A$ (filterLen $(\lambda l \to$ false) l) to $\mathbf{Vec}\ A\ 0$. The dependent casting operation that passes between provably equal indices of a dependent type (in this case $\mathbf{Vec}\ A : \mathbf{Nat} \to \mathbf{Set}$) is typically called **subst**:

$$
\begin{aligned}
&\mathbf{subst} : \{A : \mathbf{Set}\}\ \{x\ y : A\} \to (P : A \to \mathbf{Set}) \to x \equiv y \to P(x) \to P(y)\\
\\
&\mathsf{filterAll} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to \mathbf{Vec}\ A\ n \to \mathbf{Vec}\ A\ 0\\
&\mathsf{filterAll}\ \{A\}\ l = \mathbf{subst}\ (\mathbf{Vec}\ A)\ (\mathsf{lemma}\ l)\ (\mathsf{filter}\ (\lambda x \to \mathsf{false})\ l)
\end{aligned}
$$

Remark 1.1.4. The **subst** operation above is a special case of a much stronger principle stating that the two types $P(x)$ and $P(y)$ are *isomorphic* whenever $x \equiv y$: we can not only cast $P(x) \to P(y)$ but also $P(y) \to P(x)$ by symmetry of equality, and both round trips cancel. So although a proof $x \equiv y$ does not make $P(x)$ and $P(y)$ definitionally equal, they are nevertheless equal in the sense of having the same elements up to isomorphism. ◇

Uses of **subst** are very common in dependent type theory; because dependently-typed functions can both require and ensure complex invariants, one must frequently prove that the output of some function is a valid input to another.[4] Crucially, although **subst** is an "escape hatch" that compensates for the shortcomings of definitional equality, it cannot result in runtime errors—unlike explicit casts in most programming languages—because casting from $P(x)$ to $P(y)$ requires a machine-checked proof that $x \equiv y$. We can ask

[4] A more realistic variant of our lemma might account for any predicate that returns false on all the elements of the given list, not just the constantly false predicate. Alternatively, one might prove that for any $s : \mathbf{String}$, the final return type of sprintf s is $\mathbf{String}$.

for such proofs because dependent type theory is not only a functional programming language but also a higher-order intuitionistic logic that can express inductive proofs of type equality, and as we saw with filterAll, its type-checker serves also as a proof-checker.

The dependent type $x \equiv y$ is known as *propositional equality*, and it is perhaps the second most important concept in dependent type theory because it is the source of all non-definitional type equations visible within the theory. There are many formulations of propositional equality; they all implement _≡_, **refl**, and **subst** but differ in many other respects, and each has unique benefits and drawbacks. We will discuss propositional equality at length in Chapters 4 and 5.

To foreshadow the design space of propositional equality, consider that the **subst** operator may itself be subject to various definitional equalities. If we apply filterAll to a closed list *ls*, then lemma *ls* will evaluate to **refl**, so filterAll *ls* is definitionally equal to **subst** (**Vec** A) **refl** (filter ($\lambda x \to$ false) *ls*). At this point, filter ($\lambda x \to$ false) *ls* already has the desired type **Vec** A 0 because filterLen ($\lambda x \to$ false) *ls* evaluates to 0, and thus the two types involved in the cast are now definitionally equal. Ideally the **subst** term would now disappear having completed its job, and indeed the corresponding definitional equality **subst** P **refl** $x = x$ does hold for many versions of propositional equality.

Further reading

Our four categories of dependency—types/terms depending on types/terms—are reminiscent of the λ-*cube* of generalized type systems in which one augments the simply-typed λ-calculus (whose functions exhibit term-on-term dependency) with any combination of the remaining three forms of dependency [Bar91]; adding all three yields the full-spectrum dependent type theory known as the calculus of constructions [CH88]. However, the technical details of this line of work differ significantly from our presentation in Chapter 2.

The remarkable fact that type theory is both a functional programming language and a logic is known by many names including *the Curry–Howard correspondence* and *propositions as types*. It is a very broad topic with many treatments; book-length expositions include *Proofs and Types* [GLT89] and *PROGRAM = PROOF* [Mim20].

The code in this chapter is written in Agda syntax [Agda]. For more on dependently-typed programming in Agda, see *Verified Functional Programming in Agda* [Stu16]; for a more engineering-oriented perspective on dependent types, see *Type-Driven Development with Idris* [Bra17]. The sprintf example in Section 1.1.2 is inspired by the paper *Cayenne — A Language with Dependent Types* [Aug99]. Conversely, to learn about using Agda as a proof assistant for programming language theory, see *Programming Language Foundations in Agda* [WKS22].

2 Extensional type theory

In order to understand the subtle differences between modern dependent type theories, we must first study the formal definition of a dependent type theory as a mathematical object. We will then be prepared for Chapter 3, in which we study mathematical properties of type theory—and particularly of definitional and propositional equality—and their connection to computer implementations of type theory. In this chapter we therefore present the judgmental theory of Martin-Löf's *extensional type theory* [ML82], one of the canonical variants of dependent type theory. We strongly suggest following the exposition rather than simply reading the rules, but the rules are collected for convenience in Appendix A (ignoring the rules marked with (ITT), which are present only in intensional type theory).

Given the time constraints of this course, we do not attempt to give a comprehensive account of the syntax of type theories, nor do we present any of the many alternative methods of defining type theory, some of which are more efficient (but more technical) than the one we present here. These questions lead to the fascinating and deep area of *logical frameworks* which we must regrettably leave for a different course.

In this chapter In Section 2.1 we recall the concepts of judgments and inference rules in the setting of the simply-typed lambda calculus. In Section 2.2 we consider how to adapt these methods to the dependent setting, and in Section 2.3 we develop these ideas into the basic judgmental structure of dependent type theory, in which substitution plays a key role. In Section 2.4 we extend the basic rules of type theory with rules governing dependent products, dependent sums, extensional equality, and unit types. We argue that these connectives can be understood as *internalizations of judgmental structure*, a perspective which provides a conceptual justification of these connectives' rules. In Section 2.5 we define several inductive types—the empty type, booleans, and natural numbers—and explain how and why these types do not fit the pattern of the previous section. Finally, in Section 2.6 we discuss large elimination, which is implicit in our examples of full-spectrum dependency from Section 1.1, and its internalization via universe types.

Goals of the chapter By the end of this chapter, you will be able to:

- Define the core judgments of dependent type theory, and explain how and why they differ from the judgments of simple type theory.
- Explain the role of substitutions in the syntax of dependent type theory.
- Define and justify the rules of the core connectives of type theory.

2.1 *The simply-typed lambda calculus*

The theory of typed functional programming is built on extensions of a core language known as the *simply-typed lambda calculus*, which supports two types of data:

- functions of type $A \to B$ (for any types A, B): we write $\lambda x.b$ for the function that sends any input x of type A to an output b of type B, and write $f\ a$ for the application of a function f of type $A \to B$ to an input a of type A; and

- ordered pairs of type $A \times B$ (for any types A, B): we write (a, b) for the pair of a term a of type A with a term b of type B, and write $\mathbf{fst}(p)$ and $\mathbf{snd}(p)$ respectively for the first and second projections of a pair p of type $A \times B$.

It can also be seen as the implication–conjunction fragment of intuitionistic propositional logic, or as an axiom system for cartesian closed categories.

In this section we formally define the simply-typed lambda calculus as a collection of judgments presented by inference rules, in order to prepare ourselves for the analogous—but considerably more complex—definition of dependent type theory in the remainder of this chapter. Our goal is thus not to give a textbook account of the simply-typed lambda calculus but to draw the reader's attention to issues that will arise in the dependent setting.

Readers familiar with the simply-typed lambda calculus should be aware that our definition does not reference the untyped lambda calculus (as discussed in Remark 2.1.2) and considers terms modulo $\beta\eta$-equivalence (Section 2.1.2).

2.1.1 *Contexts, types, and terms*

The simply-typed lambda calculus is made up of two *sorts*, or grammatical categories, namely types and terms. We present these sorts by two well-formedness *judgments*:

- the judgment $A\ \mathsf{type}$ stating that A is a well-formed type, and

- for any well-formed type A, the judgment $a : A$ stating that a is a well-formed term of that type.

By comprehension these judgments determine respectively the collection of well-formed types and, for every element of that collection, the collection of well-formed terms of that type. (From now on we will stop writing "well-formed" because we do not consider any other kind of types or terms; see Remark 2.1.2.)

Remark 2.1.1. A judgment is simply a proposition in our ambient mathematics, one which takes part in the definition of a logical theory; we use this terminology to distinguish such meta-propositions from the propositions of the logic that is being defined [ML87].

Similarly, a sort is a type in the ambient mathematics, as distinguished from the types of the theory being defined. We refer to the ambient mathematics (in which our definition is being carried out) as the *metatheory* and the logic being defined as the *object theory*.

In this course we will be relatively agnostic about our metatheory, which the reader can imagine as "ordinary mathematics." However, one can often simplify matters by adopting a domain-specific metatheory (a *logical framework*) well-suited to defining languages/logics, as an additional level of indirection within the ambient metatheory. ⋄

Types We can easily define the types as the expressions generated by the following context-free grammar:

$$\textit{Types} \quad A, B \coloneqq \quad \mathbf{b} \mid A \times B \mid A \to B$$

We say that the judgment A type ("A is a type") holds when A is a type in the above sense. Note that in addition to function and product types we have included a base type $\mathbf{b}$; without $\mathbf{b}$ the grammar would have no terminal symbols and would thus be empty.

Equivalently, we could define the A type judgment by three *inference rules* corresponding to the three production rules in the grammar of types:

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

Each inference rule has some number of premises (here, zero or two) above the line and a single conclusion below the line; by combining these rules into trees whose leaves all have no premises, we can produce *derivations* of judgments (here, the well-formedness of a type) at the root of the tree. The tree below is a proof that $(\mathbf{b} \times \mathbf{b}) \to \mathbf{b}$ is a type:

$$\frac{\dfrac{\dfrac{}{\mathbf{b}\ \mathsf{type}} \qquad \dfrac{}{\mathbf{b}\ \mathsf{type}}}{\mathbf{b} \times \mathbf{b}\ \mathsf{type}} \qquad \dfrac{}{\mathbf{b}\ \mathsf{type}}}{(\mathbf{b} \times \mathbf{b}) \to \mathbf{b}\ \mathsf{type}}$$

Terms Terms are considerably more complex than types, so before attempting a formal definition we will briefly summarize our intentions. For the remainder of this section, fix a finite set I. The well-formed terms are as follows:

- for any $i \in I$, the base term $\mathbf{c}_i$ has type $\mathbf{b}$;
- pairing (a, b) has type $A \times B$ when $a : A$ and $b : B$;
- first projection $\mathbf{fst}(p)$ has type A when $p : A \times B$;

- second projection $\mathbf{snd}(p)$ has type B when $p : A \times B$;
- a function $\lambda x.b$ has type $A \to B$ when $b : B$ where b can contain (in addition to the usual term formers) the variable term $x : A$ standing for the function's input; and
- a function application $f\ a$ has type B when $f : A \to B$ and $a : A$.

The first difficulty we encounter is that unlike types, which are a single sort, there are infinitely many sorts of terms (one for each type) many of which refer to one another. A more significant issue is to make sense of the clause for functions: the body b of a function $\lambda x.b : A \to B$ is a term of type B according to our original grammar *extended by* a new constant $x : A$ representing an indeterminate term of type A. Because b can again be or contain a function $\lambda y.c$, we must account for finitely many extensions $x : A, y : B, \ldots$.

To account for these extensions we introduce an auxiliary sort of *contexts*, or lists of variables paired with types, representing local extensions of our theory by variable terms.

Contexts The judgment $\vdash \Gamma\ \mathsf{cx}$ ("Γ is a context") expresses that Γ is a list of pairs of term variables with types. We write $\mathbf{1}$ for the empty context and $\Gamma, x : A$ for the extension of Γ by a term variable x of type A. As a context-free grammar, we might write:

$$\begin{array}{lrl} \textit{Variables} & x, y \coloneqq & x \mid y \mid z \mid \cdots \\ \textit{Contexts} & \Gamma \coloneqq & \mathbf{1} \mid \Gamma, x : A \end{array}$$

Equivalently, in inference rule notation:

$$\frac{}{\vdash \mathbf{1}\ \mathsf{cx}} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad A\ \mathsf{type}}{\vdash \Gamma, x : A\ \mathsf{cx}}$$

We will not spend time discussing variables or binding in these lecture notes because variables will, perhaps surprisingly, not be a part of our definition of dependent type theory. For the purposes of this section we will simply assume that there is an infinite set of variables $x, y, z \ldots$, and that all the variables in any given context or term are distinct.

Terms revisited With contexts in hand we are now ready to define the term judgment, which we revise to be relative to a context Γ. The judgment $\Gamma \vdash a : A$ ("a has type A in context Γ") is defined by the following inference rules:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{i \in I}{\Gamma \vdash \mathbf{c}_i : \mathbf{b}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathbf{fst}(p) : A}$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathbf{snd}(p) : B} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

The rules for $\mathbf{c}_i$, pairing, projections, and application straightforwardly render our text into inference rule form, framed by a context Γ that is unchanged from premises to conclusion. The lambda rule explains how contexts are changed: the body of a lambda is typed in an extended context; and the variable rule explains how contexts are used: in context Γ, the variables of type A in Γ serve as additional terminal symbols of type A.

Rules such as pairing or lambda that describe how to create terms of a given type former are known as *introduction* rules, and rules describing how to use terms of a given type former, like projection and application, are known as *elimination* rules.

Remark 2.1.2. An alternative approach that is perhaps more familiar to programming languages researchers is to define a collection of *preterms*

$$\textit{Terms} \quad a, b \coloneqq \quad \mathbf{c}_i \mid x \mid (a, b) \mid \mathbf{fst}(a) \mid \mathbf{snd}(a) \mid \lambda x.a \mid a\ b$$

which includes ill-formed (typeless) terms like $\mathbf{fst}(\lambda x.x)$ in addition to the well-formed (typed) ones captured by our grammar above, and the inference rules are regarded as carving out various subsets of well-formed terms [Har16]. In fact, one often gives computational meaning to *all* preterms (as an extension of the untyped lambda calculus) and then proves that the well-typed ones are in some sense computationally well-behaved.

This is *not* the approach we are taking here; to us the term expression $\mathbf{fst}(\lambda x.x)$ does not exist any more than the type expression $\to \times \to$.[1] In fact, in light of Section 2.1.2, there will not even exist a "forgetful" map from our collections of terms to these preterms. ⋄

2.1.2 *Equational rules*

One shortcoming of our definition thus far is that our projections don't actually project anything and our function applications don't actually apply functions—there is no sense yet in which $\mathbf{fst}((a, b)) : A$ or $(\lambda x.x)\ a : A$ "are" $a : A$. Rather than equip our terms with operational meaning, we will *quotient* our terms by equations that capture a notion of sameness including these examples. The reader can imagine this process as analogous to the presentation of algebras by *generators and relations*, in which our terms thus far are the generators of a "free algebra" of (well-formed but) uninterpreted expressions.

Our true motivation for this quotient is to anticipate the definitional equality of dependent type theory, but there are certainly intrinsic reasons as well, perhaps most notably that the quotiented terms of the simply-typed lambda calculus serve as an axiom system for reasoning about cartesian closed categories [Cro94, Chapter 4].

[1] Perhaps one's definition of context-free grammar carves out the grammatical expressions out of arbitrary strings over an alphabet, but this process occurs at a different level of abstraction. The reader should banish such thoughts along with their thoughts about terms with mismatched parentheses.

We quotient by the congruence relation generated by the following rules:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathbf{fst}((a, b)) = a : A} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathbf{snd}((a, b)) = b : B} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash p = (\mathbf{fst}(p), \mathbf{snd}(p)) : A \times B}$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)\ a = b[a/x] : B} \qquad \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f = \lambda x.(f\ x) : A \to B}$$

The equations pertaining to elimination after introduction (projection from pairs and application of lambdas) are called *β-equivalences*; the equations pertaining to introduction after elimination (pairs of projections and lambdas of applications) are *η-equivalences*.

We emphasize that these equations are not *a priori* directed, and are not restricted to the "top level" of terms; we genuinely take the quotient of the collection of terms at each type by these equations, automatically inducing equations such as $\lambda x.x = \lambda x.\mathbf{fst}((x, x))$.

The first two rules explain that projecting from a pair has the evident effect. The third rule states that every term of type $A \times B$ can be written as a pair (of its projections), in effect transforming the introduction rule for products from merely a sufficient condition to a necessary one as well. Similarly, the fifth rule states that every $f : A \to B$ can be written as a lambda (of its application).

The fourth rule explains that applying a lambda function $\lambda x.b$ to an argument a is equal to the body b of that lambda with all occurrences of the placeholder variable x replaced by the term a. However, this equation makes reference to a *substitution* operation $b[a/x]$ ("substitute a for x in b") that we have not yet defined.

Substitution We can define substitution $b[a/x]$ by structural recursion on b:

$$\begin{aligned}
\mathbf{c}_i[c/x] &\coloneqq \mathbf{c}_i \\
x[c/x] &\coloneqq c \\
y[c/x] &\coloneqq y && \text{(for } x \neq y\text{)} \\
(a, b)[c/x] &\coloneqq (a[c/x], b[c/x]) \\
\mathbf{fst}(p)[c/x] &\coloneqq \mathbf{fst}(p[c/x]) \\
\mathbf{snd}(p)[c/x] &\coloneqq \mathbf{snd}(p[c/x]) \\
(\lambda y.b)[c/x] &\coloneqq \lambda y.b[c/x] && \text{(for } x \neq y\text{)} \\
(f\ a)[c/x] &\coloneqq f[c/x]\ a[c/x]
\end{aligned}$$

In the case of substituting into a lambda $(\lambda y.b)[c/x]$, we assume that the bound variable y introduced by the lambda is different from the variable x being substituted away. In practice they may coincide, in which case one must rename y (and all references to y in b)

before applying this rule. In any case, we intend this substitution to be *capture-avoiding* in the sense of not inadvertently changing the referent of bound variables.

However, because we have quotiented our collection of terms by $\beta\eta$-equivalence, it is not obvious that substitution is well-defined as a function out of the collection of terms; in order to map out of the quotient, we must check that substitution behaves equally on equal terms. (It is also not obvious that substitution is a function *into* the collection of terms, in the sense of producing well-formed terms, as we will discuss shortly.)

Consider the equation $\mathbf{fst}((a,b)) = a$. To see that substitution respects this equation, we can substitute into the left-hand side, yielding:

$$(\mathbf{fst}((a,b)))[c/x] = \mathbf{fst}((a,b)[c/x]) = \mathbf{fst}((a[c/x], b[c/x]))$$

which is β-equivalent to $a[c/x]$, the result of substituting into the right-hand side. We can check the remaining equations in a similar fashion; the $x \neq y$ condition on substitution into lambdas is necessary for substitution to respect β-equivalence of functions.

2.1.3 *Who type-checks the typing rules?*

Our stated goal in Section 2.1.1 was to define a collection of well-formed types (written $A\ \mathsf{type}$), and for each of these a collection of well-formed terms (written $a : A$). Have we succeeded? First of all, our definition of terms is now indexed by contexts Γ and written $\Gamma \vdash a : A$, to account for variables introduced by lambdas. This is no problem: we recover the original notion of (closed) term by considering the empty context $\mathbf{1}$. Nor is there any issue defining the collections of types $\mathsf{Ty} = \{A \mid A\ \mathsf{type}\}$ and contexts $\mathsf{Cx} = \{\Gamma \mid\ \vdash \Gamma\ \mathsf{cx}\}$ as presented by the grammars or inference rules in Section 2.1.1.

It is less clear that the collections of *terms* are well-defined. We would like to say that the collection of terms of type A in context Γ, $\mathsf{Tm}(\Gamma, A)$, is the set of a for which there exists a derivation of $\Gamma \vdash a : A$, modulo the relation $a \sim b \iff$ there exists a derivation of $\Gamma \vdash a = b : A$. Several questions arise immediately; for instance, is it the case that whenever $\Gamma \vdash a : A$ is derivable, Γ is a context and A is a type? If not, then we have some "junk" judgments that should not correspond to elements of some $\mathsf{Tm}(\Gamma, A)$.

Lemma 2.1.3. *If* $\Gamma \vdash a : A$ *then* $\vdash \Gamma\ \mathsf{cx}$ *and* $A\ \mathsf{type}$.

To prove such a statement, one proceeds by induction on derivations of $\Gamma \vdash a : A$. If, say, the derivation ends as follows:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash p : A \times B}}{\Gamma \vdash \mathbf{fst}(p) : A}$$

then the inductive hypothesis applied to the derivation of $\Gamma \vdash p : A \times B$ tells us that $\vdash \Gamma$ cx and $A \times B$ type. The former is exactly one of the two statements we are trying to prove. The other, A type, follows from an "inversion lemma" (proven by cases on the $-$ type judgment) that A type is not only a sufficient but also a necessary condition for $A \times B$ type.

Unfortunately our proof runs into an issue at the base cases, or at least it is not clear over what Γ the following rules range:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad\qquad \frac{i \in I}{\Gamma \vdash \mathbf{c}_i : \mathbf{b}}$$

We must either add premises to these rules stating $\vdash \Gamma$ cx, or else clarify that Γ always ranges only over contexts (which will be our strategy moving forward; see Notation 2.2.1).

Another question is the well-definedness of our quotient:

Lemma 2.1.4. *If* $\Gamma \vdash a = b : A$ *then* $\Gamma \vdash a : A$ *and* $\Gamma \vdash b : A$.

But because β-equivalence refers to substitution, proving this lemma requires:

Lemma 2.1.5 (Substitution)**.** *If* $\Gamma, x : A \vdash b : B$ *and* $\Gamma \vdash a : A$ *then* $\Gamma \vdash b[a/x] : B$.

We already saw that we must check that substitution $b[a/x]$ respects equality of b, but we must also check that it produces well-formed terms, again by induction on b. Note that substitution changes a term's context because it eliminates one of its free variables.

If we resume our attempt to prove Lemma 2.1.4, we will notice that substitution is not the only time that the context of a term changes; in the right-hand side of the η-rule of functions, f is in context $\Gamma, x : A$, whereas in the premise and left-hand side it is in Γ:

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f = \lambda x.(f\ x) : A \to B}$$

And thus we need yet another lemma.

Lemma 2.1.6 (Weakening)**.** *If* $\Gamma \vdash b : B$ *and* $\Gamma \vdash A$ type *then* $\Gamma, x : A \vdash b : B$.

We will not belabor the point any further; eventually one proves enough lemmas to conclude that we have a set of contexts Cx, a set of types Ty, and for every $\Gamma \in \mathsf{Cx}$ and $A \in \mathsf{Ty}$ a set of terms $\mathsf{Tm}(\Gamma, A)$. The complexity of each result is proportional to the complexity of that sort's definition: we define types outright, contexts by simple reference to types, and terms by more complex reference to both types and contexts. The judgments of dependent type theory are both more complex and more intertwined; rather than enduring proportionally more suffering, we will adopt a slightly different approach.

Finally, whereas all the metatheorems mentioned in this section serve only to establish that our definition is mathematically sensible, there are more genuinely interesting and contentful metatheorems one might wish to prove, including *canonicity*, the statement that (up to equality) the only closed terms of $\mathbf{b}$ are of the form $\mathbf{c}_i$ (i.e., $\mathsf{Tm}(\mathbf{1}, \mathbf{b}) = \{\mathbf{c}_i\}_{i \in I}$), and *decidability of equality*, the statement that for any $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$ we can write a program which determines whether or not $\Gamma \vdash a = b : A$.

2.2 Towards the syntax of dependent type theory

The reader is forewarned that the rules in this section serve to bridge the gap between Section 2.1 and our "official" rules for extensional type theory, which start in Section 2.3.

As we discussed in Section 1.1, the defining distinction between dependent and simple type theory is that in the former, types can contain term expressions and even term variables. Thus, whereas in Section 2.1 a simple context-free grammar sufficed to define the collection of types and we needed a context-sensitive system of inference rules to define the well-typed terms, in dependent type theory we will find that both the types and terms are context-sensitive because they refer to one another.

Types and contexts When is the dependent function type $(x : A) \to B$ well-formed? Certainly A and B must be well-formed types, but B is allowed to contain the term variable $x : A$ whereas A is not. In the case of $(n : \mathbf{Nat}) \to \mathbf{Vec}\,\mathbf{String}\,(\mathsf{suc}\,n)$, the well-formedness of the codomain depends on the fact that $\mathsf{suc}\,n$ is a well-formed term of type $\mathbf{Nat}$ (the indexing type of $\mathbf{Vec}\,\mathbf{String}$), which in turn depends on the fact that n is known to be an expression (in particular, a variable) of type $\mathbf{Nat}$.

Thus as with the *term* judgment of Section 2.1, the *type* judgment of dependent type theory must have access to the context of term variables, so we replace the A type judgment ("A is a type") of the simply-typed lambda calculus with a judgment $\Gamma \vdash A\,\mathsf{type}$ ("A is a type in context Γ"). This innocuous change has many downstream implications, so we will be fastidious about the context in which a type is well-formed.

The first consequence of this change is that contexts of term variables, which we previously defined simply as lists of well-formed types, must now also take into account *in what context* each type is well-formed. Informally we say that each type can depend on all the variables before it in the context; formally, one might define the judgment $\vdash \Gamma\,\mathsf{cx}$ by the following pair of rules:

$$\frac{}{\vdash \mathbf{1}\,\mathsf{cx}} \qquad\qquad \frac{\vdash \Gamma\,\mathsf{cx} \qquad \Gamma \vdash A\,\mathsf{type}}{\vdash \Gamma, x : A\,\mathsf{cx}}$$

Notice that the rules defining the judgment $\vdash \Gamma\,\mathsf{cx}$ refer to the judgment $\Gamma \vdash A\,\mathsf{type}$, which in turn depends on our notion of context. This kind of mutual dependence will continue to crop up throughout the rules of dependent type theory.

Notation 2.2.1 (Presuppositions)**.** With a more complex notion of context, it is more important than ever for us to decide over what Γ the judgment $\Gamma \vdash A\,\mathsf{type}$ ranges. We will say that the judgment $\Gamma \vdash A\,\mathsf{type}$ is only well-formed when $\vdash \Gamma\,\mathsf{cx}$ holds, as a matter of "meta-type discipline," and similarly that the judgment $\Gamma \vdash a : A$ is only well-formed when $\Gamma \vdash A\,\mathsf{type}$ (and thus also $\vdash \Gamma\,\mathsf{cx}$).

One often says that $\vdash \Gamma\,\mathsf{cx}$ is a *presupposition* of the judgment $\Gamma \vdash A\,\mathsf{type}$, and that the judgments $\vdash \Gamma\,\mathsf{cx}$ and $\Gamma \vdash A\,\mathsf{type}$ are presuppositions of $\Gamma \vdash a : A$. We will globally adopt the convention that whenever we assert the truth of some judgment in prose or as the premise of a rule, we also implicitly assert that its presuppositions hold. Dually, we will be careful to check that none of our rules have meta-ill-typed conclusions.

Now that we have added a term variable context to the type well-formedness judgment, we can explain when $(x : A) \to B$ is a type: it is a (well-formed) type in Γ when A is a type in Γ and B is a type in $\Gamma, x : A$, as follows.

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma, x : A \vdash B\,\mathsf{type}}{\Gamma \vdash (x : A) \to B\,\mathsf{type}}$$

Rules like this describing how to create a type are known as *formation rules*, to parallel the terminology of introduction and elimination rules.

We can now sketch the formation rules for many of the types we encountered in Chapter 1. Dependent types like _≡_ and **Vec** are particularly interesting because they entangle the $\Gamma \vdash A\,\mathsf{type}$ judgment with the term well-formedness judgment $\Gamma \vdash a : A$.

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash \mathbf{Nat}\,\mathsf{type}} \qquad \frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{Vec}\,A\,n\,\mathsf{type}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \equiv b\,\mathsf{type}}$$

Note that the convention of presuppositions outlined in Notation 2.2.1 means that the second and third rules have an implicit $\vdash \Gamma\,\mathsf{cx}$ premise, and the third rule also has an implicit $\Gamma \vdash A\,\mathsf{type}$ premise. To see that the conclusions of these rules are meta-well-typed, we must check that $\vdash \Gamma\,\mathsf{cx}$ holds in each case; this is an explicit premise of the first rule and a presupposition of the premises of the second and third rules.

The formation rule for propositional equality _≡_ in particular is a major source of dependency because it singlehandledly allows arbitrary terms of arbitrary type to occur within types. In fact, this rule by itself causes the inference rules of all three judgments $\vdash \Gamma\,\mathsf{cx}$, $\Gamma \vdash A\,\mathsf{type}$, and $\Gamma \vdash a : A$ to all depend on one another pairwise.

Exercise 2.1. *Attempt to derive that* $(n : \mathbf{Nat}) \to \mathbf{Vec}\,\mathbf{String}\,(\mathsf{suc}\,n)$ *is a well-formed type in the empty context* $\mathbf{1}$*, using the rules introduced in this section thus far. Several rules are missing; which judgments can you not yet derive?*

The variable rule Let us turn now to the term judgment $\Gamma \vdash a : A$, and in particular the rule stating that term variables in the context are well-formed terms. For simplicity, imagine the special case where the last variable is the one under consideration:

$$\frac{}{\Gamma, x : A \vdash x : A}\ !?$$

This rule needs considerable work, as neither of the conclusion's presuppositions, $\vdash (\Gamma, x : A)\ \mathsf{cx}$ and $\Gamma, x : A \vdash A\ \mathsf{type}$, currently hold. We can address the former by adding premises $\vdash \Gamma\ \mathsf{cx}$ and $\Gamma \vdash A\ \mathsf{type}$ to the rule, from which it follows that $\vdash (\Gamma, x : A)\ \mathsf{cx}$.[2] As for the latter, note that $\Gamma \vdash A\ \mathsf{type}$ does not actually imply $\Gamma, x : A \vdash A\ \mathsf{type}$—this would require proving a *weakening lemma* (see Lemma 2.1.6) for types! (Conversely, if the rule has the premise $\Gamma \vdash A\ \mathsf{type}$, then we cannot establish well-formedness of the context.)

There are several ways to proceed. One is to prove a weakening lemma, but given that the well-formedness of the variable rule requires weakening, it is necessary to prove all our well-formedness, weakening, and substitution lemmas by a rather heavy simultaneous induction. A second approach would be to add a silent weakening *rule* stating that $\Gamma, x : A \vdash B\ \mathsf{type}$ whenever $\Gamma \vdash B\ \mathsf{type}$; however, this introduces ambiguity into our rules regarding the context(s) in which a type or term is well-formed.

We opt for a third option, which is to add *explicit* weakening rules asserting the existence of an operation sending types and terms in context Γ to types and terms in context $\Gamma, x : A$, both written $-[\mathbf{p}]$. (This notation will become less mysterious later.)

$$\frac{\Gamma \vdash B\ \mathsf{type} \qquad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A \vdash B[\mathbf{p}]\ \mathsf{type}} \qquad\qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A \vdash b[\mathbf{p}] : B[\mathbf{p}]}$$

Note that the type weakening rule is needed to make sense of the term weakening rule.

We can now fix the variable rule we wrote above: using $-[\mathbf{p}]$ to weaken A by itself, we move A from context Γ to $\Gamma, x : A$ as required in the conclusion of the rule.

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A \vdash x : A[\mathbf{p}]}$$

To use variables that occur earlier in the context, we can apply weakening repeatedly until they are the last variable. Suppose that $\mathbf{1} \vdash A\ \mathsf{type}$ and $x : A \vdash B\ \mathsf{type}$, and in the context $x : A, y : B$ we want to use the variable x. Ignoring the $y : B$ in the context for a moment, we know that $x : A \vdash x : A[\mathbf{p}]$ by the last variable rule; thus by weakening we

[2] Of course one could just directly add the premise $\vdash (\Gamma, x : A)\ \mathsf{cx}$, but our short-term memory is robust enough to recall that our next task is to ensure that A is a type.

have $x : A, y : B \vdash x[\mathbf{p}] : A[\mathbf{p}][\mathbf{p}]$. In general, we can derive the following principle:

$$\frac{\Gamma \vdash A \,\mathsf{type} \qquad \Gamma, x : A \vdash B_1 \,\mathsf{type} \qquad \dots \qquad \Gamma, x : A, y_1 : B_1, \dots \vdash B_n \,\mathsf{type}}{\Gamma, x : A, y_1 : B_1, \dots, y_n : B_n \vdash x\underbrace{[\mathbf{p}] \dots [\mathbf{p}]}_{n \text{ times}} : A\underbrace{[\mathbf{p}] \dots [\mathbf{p}]}_{n+1 \text{ times}}}$$

This approach to variables is elegant in that it breaks the standard variable rule into two simpler primitives: a rule for the last variable, and rules for type and term weakening. However, it introduces a redundancy in our notation, because the term $x[\mathbf{p}]^n$ encodes in two different ways the variable to which it refers: by the name x as well as positionally by the number of weakenings n.

A happy accident of our presentation of the variable rule is thus that we can delete variable names altogether; in Section 2.3 we will present contexts simply as lists of types $A.B.C$ with no variable names, and adopt a single notation for "the last variable in the context," an encoding of the lambda calculus known as *de Bruijn indexing* [Bru72]. Conceptual elegance notwithstanding, this notation is very unfriendly to the reader in larger examples[3] so we will continue to use named variables outside of the rules themselves; translating between the two notations is purely mechanical.

Remark 2.2.2. The first author wishes to mention another approach to maintaining readability, which is to continue using both named variables and explicit weakenings [Gra09]; this approach has the downside of requiring us to explain variable binding, but is simultaneously readable and precise about weakenings. ⋄

2.3 *The calculus of substitutions*

Weakening is one of two main operations in type theory that moves types and terms between contexts, the other being substitution of terms for variables. For the same reasons that we want to present weakening as an explicit type- and term-forming operation, we will also formulate substitution as an explicit operation subject to equations explicating how it computes on each construct of the theory.

However, rather than axiomatizing *single* substitutions and weakenings, we will axiomatize arbitrary compositions of substitutions and weakenings. In light of the fact that substitution shortens the context of a type/term and weakening lengthens it, these composite operations—called *simultaneous substitutions* (henceforth just substitutions)—can turn any context Γ into any other context Δ.

[3] According to Conor McBride, "Bob Atkey once memorably described the capacity to put up with de Bruijn indices as a Cylon detector." (`https://mazzo.li/epilogue/index.html%3Fp=773.html`)

We thus add one final judgment to our presentation of type theory, $\Delta \vdash \gamma : \Gamma$ ("γ is a substitution from Δ to Γ"), corresponding to operations that send types/terms from context Γ to context Δ. (Not a typo; we will address the "backwards" notation later.)

Notation 2.3.1. Type theory has four basic judgments and three equality judgments:

1. $\vdash \Gamma\,\mathsf{cx}$ asserts that Γ is a context.
2. $\Delta \vdash \gamma : \Gamma$, presupposing $\vdash \Delta\,\mathsf{cx}$ and $\vdash \Gamma\,\mathsf{cx}$, asserts that γ is a substitution from Δ to Γ.
3. $\Gamma \vdash A\,\mathsf{type}$, presupposing $\vdash \Gamma\,\mathsf{cx}$, asserts that A is a type in context Γ.
4. $\Gamma \vdash a : A$, presupposing $\vdash \Gamma\,\mathsf{cx}$ and $\Gamma \vdash A\,\mathsf{type}$, asserts that a is an element/term of type A in context Γ.

2′. $\Delta \vdash \gamma = \gamma' : \Gamma$, presupposing $\Delta \vdash \gamma : \Gamma$ and $\Delta \vdash \gamma' : \Gamma$, asserts that γ, γ' are equal substitutions from Δ to Γ.

3′. $\Gamma \vdash A = A'\,\mathsf{type}$, presupposing $\Gamma \vdash A\,\mathsf{type}$ and $\Gamma \vdash A'\,\mathsf{type}$, asserts that A, A' are equal types in context Γ.

4′. $\Gamma \vdash a = a' : A$, presupposing $\Gamma \vdash a : A$ and $\Gamma \vdash a' : A$, asserts that a, a' are equal elements of type A in context Γ.

Notation 2.3.2. We write Cx for the set of contexts, $\mathsf{Sb}(\Delta, \Gamma)$ for the set of substitutions from Δ to Γ, $\mathsf{Ty}(\Gamma)$ for the set of types in context Γ, and $\mathsf{Tm}(\Gamma, A)$ for the set of terms of type A in context Γ.

This presentation of dependent type theory is known as the *substitution calculus* [ML92; Tas93]. Perhaps unsurprisingly, we must discuss a considerable number of rules governing substitutions before presenting any concrete type and term formers; we devote this section to those rules, and cover the main connectives of type theory in Section 2.4.

Contexts The rules for contexts are as in Section 2.2, but without variable names:

$$\frac{}{\vdash \mathbf{1}\,\mathsf{cx}} \qquad\qquad \frac{\vdash \Gamma\,\mathsf{cx} \qquad \Gamma \vdash A\,\mathsf{type}}{\vdash \Gamma.A\,\mathsf{cx}}$$

Although there is no context equality judgment, note that two contexts *can* be equal without being syntactically identical. If $\mathbf{1} \vdash A = A'\,\mathsf{type}$ then $\mathbf{1}.A$ and $\mathbf{1}.A'$ are equal contexts on the basis that, like all operations of the theory, context extension respects equality in both arguments. We have omitted the $\vdash \Gamma = \Gamma'\,\mathsf{cx}$ judgment for the simple reason that there would be no rules governing it: the only reason why two contexts can be equal is that their types are pairwise equal.

Substitutions The purpose of a substitution $\Delta \vdash \gamma : \Gamma$ is to shift types and terms from context Γ to context Δ:

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type}}{\Delta \vdash A[\gamma]\,\mathsf{type}} \qquad\qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash a[\gamma] : A[\gamma]}$$

Unlike the substitution operation of Section 2.1, which was a function on terms defined by cases, these rules define two binary type- and term- forming operations that take a type (resp., term) and a substitution as input and produce a new type (resp., term). Note also that, despite sharing a notation, type and term substitution are two distinct operations.

The simplest interesting substitution is weakening, written $\mathbf{p}$:[4]

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma.A \vdash \mathbf{p} : \Gamma}$$

In concert with the substitution rules above we can recover the weakening rules from the previous section, e.g., if $\Gamma \vdash B\,\mathsf{type}$ and $\Gamma \vdash A\,\mathsf{type}$ then $\Gamma, x : A \vdash B[\mathbf{p}]\,\mathsf{type}$.

Because substitutions $\Delta \vdash \gamma : \Gamma$ encode arbitrary compositions of context-shifting operations, we also have rules that close substitutions under nullary and binary composition:

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash \mathbf{id} : \Gamma} \qquad\qquad \frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_2 \vdash \gamma_0 \circ \gamma_1 : \Gamma_0}$$

These operations are unital and associative as one might expect:

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \gamma \circ \mathbf{id} = \mathbf{id} \circ \gamma = \gamma : \Gamma} \qquad\qquad \frac{\Gamma_3 \vdash \gamma_2 : \Gamma_2 \qquad \Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_3 \vdash \gamma_0 \circ (\gamma_1 \circ \gamma_2) = (\gamma_0 \circ \gamma_1) \circ \gamma_2 : \Gamma_0}$$

We can summarize the rules above by stating that there is a *category* whose objects are contexts and whose morphisms are substitutions.

We have already seen that substitutions shift the contexts of types and terms by $-[\gamma]$; they also shift the context of other substitutions by precomposition. Later we will have occasion to discuss all three context-shifting functions between sorts that are induced by substitutions, as follows.

Notation 2.3.3. Given a substitution $\Delta \vdash \gamma : \Gamma$, we write γ^* for the following functions:

- $\xi \mapsto \xi \circ \gamma : \mathsf{Sb}(\Gamma, \Xi) \to \mathsf{Sb}(\Delta, \Xi)$,
- $A \mapsto A[\gamma] : \mathsf{Ty}(\Gamma) \to \mathsf{Ty}(\Delta)$, and

[4] This mysterious name can be explained by the fact that weakening corresponds semantically to a projection map; $\mathbf{p}$ can thus be pronounced as either "weakening" or "projection".

- $a \mapsto a[\gamma] : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}(\Delta, A[\gamma])$.

Composite substitutions introduce a possible redundancy into our rules: what is the difference between substituting by γ_0 and then by γ_1 versus substituting once by $\gamma_0 \circ \gamma_1$? We add equations asserting that substituting by **id** is the identity and substituting by a composite is composition of substitutions:

$$\frac{\Gamma \vdash A \ \mathsf{type}}{\Gamma \vdash A[\mathbf{id}] = A \ \mathsf{type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash a[\mathbf{id}] = a : A}$$

$$\frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash A \ \mathsf{type}}{\Gamma_2 \vdash A[\gamma_0 \circ \gamma_1] = A[\gamma_0][\gamma_1] \ \mathsf{type}} \qquad \frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash a : A}{\Gamma_2 \vdash a[\gamma_0 \circ \gamma_1] = a[\gamma_0][\gamma_1] : A[\gamma_0 \circ \gamma_1]}$$

We can summarize the rules above by stating that the γ^* operations respect identity and composition of substitutions, or more compactly, that the collections of types and terms form *presheaves* $\mathsf{Ty}(-)$ and $\sum_{A:\mathsf{Ty}(-)} \mathsf{Tm}(-, A)$ on the category of contexts, with restriction maps given by substitution (a perspective which inspires the notation γ^*).

Before moving on, it is instructive to once again convince ourselves that the rules above are meta-well-typed. In particular, the conclusion of the second rule is only sensible if $\Gamma \vdash a[\mathbf{id}] : A$, but according to the rule for term substitution we only have $\Gamma \vdash a[\mathbf{id}] : A[\mathbf{id}]$. To make sense of this rule we must refer to the previous rule equating the types $A[\mathbf{id}]$ and A. A consequence of this type equation is that terms of type $A[\mathbf{id}]$ are equivalently terms of type A,[5] and thus $\Gamma \vdash a[\mathbf{id}] : A$ as required. This is a paradigmatic example of the deeply intertwined nature of the rules of dependent type theory; in particular, *we cannot defer equations* to the end of our construction the way we did in Section 2.1 because many rules are only sensible after imposing certain equations.

The variable rule revisited As in the previous section, the variable rule is restricted to the last entry in the context, which we (unambiguously) always name **q**.[6]

$$\frac{\Gamma \vdash A \ \mathsf{type}}{\Gamma.A \vdash \mathbf{q} : A[\mathbf{p}]}$$

Writing $\mathbf{p}^n$ for the n-fold composition of $\mathbf{p}$ with itself (with $\mathbf{p}^0 = \mathbf{id}$), the following rule is *derivable* from other rules (notated $\Rightarrow$) and thus not explicitly included in our system:

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma.A \vdash B_1 \ \mathsf{type} \qquad \dots \qquad \Gamma.A.B_1 \dots \vdash B_n \ \mathsf{type}}{\Gamma.A.B_1 \dots B_n \vdash \mathbf{q}[\mathbf{p}^n] : A[\mathbf{p}^{n+1}]} \Rightarrow$$

[5] In some presentations of type theory this principle is explicit and is known as the *type conversion rule*. For us it is a consequence of the judgments respecting equality, i.e., $\mathsf{Tm}(\Gamma, A[\mathbf{id}]) = \mathsf{Tm}(\Gamma, A)$ as sets.

[6] This mysterious name is chosen to pair well with the name **p** that we gave weakening; **q** can thus be pronounced as either "variable" or "qariable".

Thus a variable in our system is a term of the form $\mathbf{q}[\mathbf{p}^n]$, where n is its de Bruijn index.

Terminal substitutions Our notation $\Delta \vdash \gamma : \Gamma$ for substitutions is no accident; it is indeed a good mental model to think of such substitutions as "terms of type Γ in context Δ." To understand why, let us think back to propositional logic. A term $\mathbf{1}.B \vdash c : C$ can be seen as a proof of C under the hypothesis B, i.e., a proof that $B \implies C$. Given a substitution $\mathbf{1}.A \vdash b : \mathbf{1}.B$ we can obtain a term $\mathbf{1}.A \vdash c[b] : C[b]$, or a proof that $A \implies C$. This suggests that substituting corresponds logically to a "cut," and b to a proof that $A \implies B$.

Returning to the general case, contexts are lists of hypotheses, and a substitution $\Delta \vdash \gamma : \Gamma$ states that we can prove all the hypotheses of Γ using the hypotheses of Δ. Thus anything that is true under the hypotheses Γ is also true under the hypotheses Δ—hence the contravariance of the substitution operation.

More concretely, the idea is that a substitution $\Delta \vdash \gamma : \mathbf{1}.A_1 \dots A_n$ is an n-tuple of terms $a_1, \dots, a_n$ of types $A_1, \dots, A_n$, all in context Δ, and applying the substitution γ has the effect of substituting a_1 for the first variable, a_2 for the second variable, …and a_n for the last variable. The final subtlety is that each type A_i is in general dependent on all the previous A_j for $j < i$, so the type of a_2 is not just A_2 but "$A_2[a_1/x_1]$," so to speak, all the way through "$a_n : A_n[a_1/x_1, \dots, a_{n-1}/x_{n-1}]$."

If all of this sounds very complicated, well…at any rate, the remaining rules governing substitution define such n-tuples in two cases, 0 and $n+1$. The nullary case is fairly simple: any substitution $\Gamma \vdash \delta : \mathbf{1}$ into the empty context (a length-zero list of types) is necessarily the empty tuple $\langle\rangle$, which we spell !.

$$\frac{\vdash \Gamma \ \mathsf{cx}}{\Gamma \vdash \,! : \mathbf{1}} \qquad\qquad \frac{\Gamma \vdash \delta : \mathbf{1}}{\Gamma \vdash \,! = \delta : \mathbf{1}}$$

These rules state that $\mathbf{1}$ is a terminal object in the category of contexts, a perspective which inspires the notations $\mathbf{1}$ and !.

Substitution extension The other case concerns substitutions $\Delta \vdash - : \Gamma.A$ into a context extension. Recall that $\Gamma.A$ is an $(n+1)$-tuple of types when Γ is an n-tuple of types, and suppose that $\Delta \vdash \gamma : \Gamma$, which is to say that γ is an n-tuple of terms (in context Δ) whose types are those in Γ. To extend this n-tuple to an $(n+1)$-tuple of terms whose types are those in $\Gamma.A$, we simply adjoin one more term a in context Δ with type $A[\gamma]$, where this substitution plugs the n previously-given terms into the dependencies of A.

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A \ \mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \gamma.a : \Gamma.A}$$

The final three rules of our calculus are equations governing this substitution former:

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A \,\mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{p} \circ (\gamma.a) = \gamma : \Gamma} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A \,\mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{q}[\gamma.a] = a : A[\gamma]}$$

$$\frac{\Gamma \vdash A \,\mathsf{type} \qquad \Delta \vdash \gamma : \Gamma.A}{\Delta \vdash \gamma = (\mathbf{p} \circ \gamma).\mathbf{q}[\gamma] : \Gamma.A}$$

Imagining for the moment that $\Gamma = x_1 : A_1, \dots, x_n : A_n$ and $\gamma = [a_1/x_1, \dots, a_n/x_n]$, the second rule states that $x_n[a_1/x_1, \dots, a_n/x_n] = a_n$, in other words, that substituting into the last variable x_n replaces that variable by the last term a_n. The first rule states in essence that substituting into a type/term that does not mention (is weakened by) x_n is the same as dropping the last term a_n/x_n from the substitution, i.e., $[a_1/x_1, \dots, a_{n-1}/x_{n-1}]$.

Finally, the third rule states that every substitution γ into the context $\Gamma.A$ is of the form $\gamma_0.a$, where a is determined by the behavior of γ on the last variable, and γ_0 is determined by the behavior of γ on the first n variables. (See Exercise 2.5.)

All of these rules in this section determine a category (of contexts and substitutions) with extra structure, known collectively as a *category with families* [Dyb96]. We will refer to any system that extends this collection of rules as a *Martin-Löf type theory.*

Exercise 2.2. *Show that substitutions* $\Gamma \vdash \gamma : \Gamma.A$ *satisfying* $\mathbf{p} \circ \gamma = \mathbf{id}$ *are in bijection with terms* $\Gamma \vdash a : A$.

Exercise 2.3. *Show that* $(\gamma.a) \circ \delta = (\gamma \circ \delta).a[\delta]$.

Exercise 2.4. *Given* $\Delta \vdash \gamma : \Gamma$ *and* $\Gamma \vdash A \,\mathsf{type}$*, construct a substitution that we will name* $\gamma.A$*, satisfying* $\Delta.A[\gamma] \vdash \gamma.A : \Gamma.A$.

Exercise 2.5. *Suppose that* $\Gamma \vdash A \,\mathsf{type}$ *and* $\vdash \Delta \,\mathsf{cx}$. *Show that substitutions* $\Delta \vdash \gamma : \Gamma.A$ *are in bijection with pairs of a substitution* $\Delta \vdash \gamma_0 : \Gamma$ *and a term* $\Delta \vdash a : A[\gamma_0]$.

2.4 *Internalizing judgmental structure:* Π, Σ, Eq, Unit

With the basic structure of dependent type theory finally out of the way, we are prepared to define standard type and term formers, starting with the best-behaved connectives: dependent products, dependent sums, extensional equality, and the unit type. Unlike inductive types (Section 2.5), each of these connectives can be described concisely as internalizing judgmental structure of some kind.

2.4.1 *Dependent products*

We start with dependent function types, also known as *dependent products* or $\mathbf{\Pi}$-*types*. The formation rule is as in Section 2.2, but without variable names:[7]

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma.A \vdash B \text{ type}}{\Gamma \vdash \mathbf{\Pi}(A, B) \text{ type}}$$

Remark 2.4.1. The $\mathbf{\Pi}$ notation and terminology is inspired by this type corresponding semantically to a set-indexed product of sets $\prod_{a \in A} B_a$. Indexed products generalize ordinary products in the sense that $\prod_{a \in \{1,2\}} B_a \cong B_1 \times B_2$. ⋄

Remarkably, the substitution calculus ensures that these rules are almost indistinguishable from the introduction and elimination rules of simple function types in Section 2.1, with some minor additional bookkeeping to move types to the appropriate contexts:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \lambda(b) : \mathbf{\Pi}(A, B)} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B \text{ type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A, B)}{\Gamma \vdash \mathbf{app}(f, a) : B[\mathbf{id}.a]}$$

There continue to be a few notational shifts: λs no longer come with variable names, and we write $\mathbf{app}(f, a)$ rather than $f\ a$ just to emphasize that function application is a term constructor. The reader should convince themselves that in the final rule, $\Gamma \vdash B[\mathbf{id}.a]$ type; this substitutes a for the last variable in B, leaving the rest of the context unchanged.

Next we must specify equations not only on the introduction and elimination forms, but on the type former itself. There are two groups of equations we must impose; the first group explains how substitutions act on all three of these operations:

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A \text{ type} \qquad \Gamma.A \vdash B \text{ type}}{\Delta \vdash \mathbf{\Pi}(A, B)[\gamma] = \mathbf{\Pi}(A[\gamma], B[\gamma.A]) \text{ type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A \text{ type} \qquad \Gamma.A \vdash b : B}{\Delta \vdash \lambda(b)[\gamma] = \lambda(b[\gamma.A]) : \mathbf{\Pi}(A, B)[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B \text{ type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A, B)}{\Delta \vdash \mathbf{app}(f, a)[\gamma] = \mathbf{app}(f[\gamma], a[\gamma]) : B[\gamma.a[\gamma]]}$$

Roughly speaking, these three rules state that substitutions commute past each type and term former, but B and b are well-formed in a larger context ($\Gamma.A$) than the surrounding term (Γ), requiring us to "shift" the substitution so that it leaves the bound variable of type A unchanged while continuing to act on all the free variables in Γ. (The "shifted" substitution $\gamma.A$ in these rules is the derived form defined in Exercise 2.4.)

Once again we should pause and convince ourselves that these rules are meta-well-typed. Echoing the phenomenon we saw in Section 2.3 with $\Gamma \vdash a[\mathbf{id}] : A$, we need to

[7] We have switched our notation from $(x : A) \to B$ because it is awkward without named variables.

use the substitution rule for $\mathbf{\Pi}(A,B)[\gamma]$ to see that the right-hand side of the substitution rules for $\lambda(b)[\gamma]$ and $\mathbf{app}(f,a)[\gamma]$ are well-typed.

Exercise 2.6. *Check that the substitution rule for* $\mathbf{app}(f,a)[\gamma]$ *is meta-well-typed; in particular, show that both* $\mathbf{app}(f,a)[\gamma]$ *and* $\mathbf{app}(f[\gamma],a[\gamma])$ *have the type* $B[\gamma.a[\gamma]]$.

This pattern will continue: every time we introduce a new type or term former θ, we will add an equation $\theta(a_1,\ldots,a_n)[\gamma] = \theta(a_1[\gamma_1],\ldots,a_n[\gamma_n])$ stating that substitutions push past θ, adjusted as necessary in each argument. These rules are quite mechanical and can even be automatically derived in some frameworks, but they are at the heart of type theory itself. From a logical perspective, they ensure that quantifier instantiation is uniform. From a mathematical perspective, as we will see in Section 2.4.2, they assert the naturality of type-theoretic constructions. And from an implementation perspective, these rules can be assembled into a substitution algorithm, ensuring that substitutions can be computed automatically by proof assistants.

Remark 2.4.2. The difference between this approach to substitution and the one outlined in Section 2.1 is one of *derivability* vs *admissibility*. In the simply-typed setting, the fact that all terms enjoy substitution is not part of the system but rather must be proven (and even constructed in the first place) by induction over the structure of terms, and so adding new constructs to the theory may cause substitution to fail.

In the substitution calculus, we assert that all types and terms enjoy substitution as basic rules of the theory, and later add equations specifying how substitution computes; thus any extension of the theory is guaranteed to enjoy substitution. Because substitution is a crucial aspect of dependent type theory, we find this latter approach more ergonomic. ⋄

The second group of equations is the β- and η-rules introduced in Section 2.1, completing our presentation of dependent product types.

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \mathbf{app}(\lambda(b),a) = b[\mathbf{id}.a] : B[\mathbf{id}.a]} \qquad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Gamma \vdash f = \lambda(\mathbf{app}(f[\mathbf{p}],\mathbf{q})) : \mathbf{\Pi}(A,B)}$$

Exercise 2.7. *Carefully explain why the η-rule above is meta-well-typed, in particular why* $\lambda(\mathbf{app}(f[\mathbf{p}],\mathbf{q}))$ *has the right type. Explicitly point out all the other rules and equations (e.g.,* **Π***-introduction,* **Π***-elimination, weakening) to which you refer.*

Exercise 2.8. *Show that using* **Π***-types we can define a non-dependent function type whose formation rule states that if* $\Gamma \vdash A\ \mathsf{type}$ *and* $\Gamma \vdash B\ \mathsf{type}$ *then* $\Gamma \vdash A \to B\ \mathsf{type}$. *Then define the introduction and elimination rules from Section 2.1 for this encoding, and check that the β- and η-rules from Section 2.1 hold. (Hint: it is incorrect to define* $A \to B :\equiv \mathbf{\Pi}(A,B)$*.)*

Exercise 2.9. *As discussed in Section 2.3, two contexts that are not syntactically identical may nevertheless be equal. Give an example.*

2.4.2 *Dependent products internalize hypothetical judgments*

With one type constructor, two term constructors, and five equations, it is natural to wonder whether we have written "enough" or "the correct" rules to specify $\mathbf{\Pi}$-types. One may also wonder whether there is an easier way. We now introduce a methodology for making sense of this collection of rules, and show how we can use this methodology to more efficiently define the later connectives. In short, we will view connectives as *internalizations of judgmental structure*, and $\Gamma \vdash - : \mathbf{\Pi}(A, B)$ in particular as an internalization of the hypothetical judgment $\Gamma.A \vdash - : B$.

Remark 2.4.3. In these notes we limit ourselves to a semi-informal discussion of this perspective, which can be made fully precise with the language of category theory. For instance, using the framework of natural models, Awodey [Awo18] shows that the rules above exactly capture that $\mathbf{\Pi}$-types classify the hypothetical judgment in a precise sense. ⋄

Analyzing context extension To warm up, let us begin by recalling Exercise 2.5, which establishes the following bijection of sets for every Δ, Γ, and A:

$$\{\gamma \mid \Delta \vdash \gamma : \Gamma.A\} \cong \{(\gamma_0, a) \mid \Delta \vdash \gamma_0 : \Gamma \wedge \Delta \vdash a : A[\gamma_0]\}$$

Using Notation 2.3.2 we equivalently write:

$$\iota_{\Delta,\Gamma,A} : \mathsf{Sb}(\Delta, \Gamma.A) \cong \textstyle\sum_{\gamma \in \mathsf{Sb}(\Delta,\Gamma)} \mathsf{Tm}(\Delta, A[\gamma])$$

where $\sum_{a \in A} B_a$ is our notation for the set-indexed coproduct of sets $\coprod_{a \in A} B_a$.

As stated, the bijections $\iota_{\Delta,\Gamma,A}$ and $\iota_{\Delta',\Gamma',A'}$ may be totally unrelated, but it turns out that this collection of bijections is actually *natural* (or "parametric") in Δ in the sense that the behavior of $\iota_{\Delta_0,\Gamma,A}$ and $\iota_{\Delta_1,\Gamma,A}$ are correlated when we have a substitution from Δ_0 to Δ_1.

Because these bijections have different types, to make this idea precise we must find a way to relate their differing domains $\mathsf{Sb}(\Delta_0, \Gamma.A)$ and $\mathsf{Sb}(\Delta_1, \Gamma.A)$ with one another, as well as their codomains $\sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])$ and $\sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma])$.

We have already seen the former in Notation 2.3.3: every substitution $\Delta_0 \vdash \delta : \Delta_1$ induces a function $\delta^* : \mathsf{Sb}(\Delta_1, \Gamma.A) \to \mathsf{Sb}(\Delta_0, \Gamma.A)$. We leave the latter as an exercise:

Exercise 2.10. *Given* $\Delta_0 \vdash \delta : \Delta_1$*, use* δ^* *(Notation 2.3.3) to define the following function:*

$$\textstyle\sum_{\delta^*} \delta^* : \sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma]) \to \sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])$$

Proof. Define $(\sum_{\delta^*} \delta^*)(\gamma, a) = (\delta^* \gamma, \delta^* a) = (\gamma \circ \delta, a[\delta])$. □

With these functions in hand we can now explain precisely what we mean by the naturality of $\iota_{-,\Gamma,A}$. Fix a substitution $\Delta_0 \vdash \delta : \Delta_1$. We have two different ways of turning a substitution $\Delta_1 \vdash \gamma : \Gamma.A$ into an element of $\sum_{\gamma_0 \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma_0])$, depicted by the "right then down" and "down then right" paths in the diagram below:

$$\begin{tikzcd}
\mathsf{Sb}(\Delta_1, \Gamma.A) \arrow[r, "\iota_{\Delta_1,\Gamma,A}"] \arrow[d, "\delta^*"'] & \sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma]) \arrow[d, "\sum_{\delta^*} \delta^*"] \\
\mathsf{Sb}(\Delta_0, \Gamma.A) \arrow[r, "\iota_{\Delta_0,\Gamma,A}"'] & \sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])
\end{tikzcd}$$

Going "right then down" we obtain

$$\begin{tikzcd}
\gamma \arrow[r, mapsto] & \iota_{\Delta_1,\Gamma,A}(\gamma) \arrow[d, mapsto] \\
& (\sum_{\delta^*} \delta^*)(\iota_{\Delta_1,\Gamma,A}(\gamma))
\end{tikzcd}$$

and going "down then right" we obtain $\gamma \mapsto \gamma \circ \delta \mapsto \iota_{\Delta_0,\Gamma,A}(\gamma \circ \delta)$.

We say that the family of isomorphisms $\Delta \mapsto \iota_{\Delta,\Gamma,A}$ is natural when these two paths always yield the same result, i.e., when $(\sum_{\delta^*} \delta^*)(\iota_{\Delta_1,\Gamma,A}(\gamma)) = \iota_{\Delta_0,\Gamma,A}(\gamma \circ \delta)$ for every $\Delta_0 \vdash \delta : \Delta_1$ and γ. In other words, $\iota_{\Delta_0,\Gamma,A}$ and $\iota_{\Delta_1,\Gamma,A}$ "do the same thing" as soon as you correct the mismatch in their types by pre- and post-composing the appropriate maps.

Exercise 2.11. *Prove that ι is natural, i.e., that the following maps are equal:*

$$\textstyle\sum_{\delta^*} \delta^* \circ \iota_{\Delta_1,\Gamma,A} = \iota_{\Delta_0,\Gamma,A} \circ \delta^* : \mathsf{Sb}(\Delta_1, \Gamma.A) \to \sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])$$

Proof. Suppose $\gamma \in \mathsf{Sb}(\Delta_1, \Gamma.A)$. Unfolding the solutions to Exercises 2.5 and 2.10,

$$\begin{gathered}
(\textstyle\sum_{\delta^*} \delta^*)(\iota_{\Delta_1,\Gamma,A}(\gamma)) = (\sum_{\delta^*} \delta^*)(\mathbf{p} \circ \gamma, \mathbf{q}[\gamma]) = ((\mathbf{p} \circ \gamma) \circ \delta, \mathbf{q}[\gamma][\delta]) \\
\iota_{\Delta_0,\Gamma,A}(\delta^*(\gamma)) = \iota_{\Delta_0,\Gamma,A}(\gamma \circ \delta) = (\mathbf{p} \circ (\gamma \circ \delta), \mathbf{q}[\gamma \circ \delta])
\end{gathered}$$

which are equal by the functoriality of substitution. □

The terminology of "natural" comes from category theory, where $\iota_{-,\Gamma,A}$ is known as a natural isomorphism, but we will prove and use naturality conditions without referring to the general concept. One useful consequence of naturality is the following:

Exercise 2.12. Without *unfolding the definition of ι, show that the naturality of ι and the fact that $\iota_{\Delta,\Gamma,A}$ and $\iota^{-1}_{\Delta,\Gamma,A}$ are inverses together imply that ι^{-1} is natural, i.e., that*

$$\iota^{-1}_{\Delta_0,\Gamma,A} \circ \textstyle\sum_{\delta^*} \delta^* = \delta^* \circ \iota^{-1}_{\Delta_1,\Gamma,A} : \sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma]) \to \mathsf{Sb}(\Delta_0, \Gamma.A)$$

Proof. Apply $\iota^{-1}_{\Delta_0,\Gamma,A} \circ - \circ \iota^{-1}_{\Delta_1,\Gamma,A}$ to both sides of the naturality equation for ι and cancel:

$$\iota^{-1}_{\Delta_0,\Gamma,A} \circ \textstyle\sum_{\delta^*} \delta^* \circ \iota_{\Delta_1,\Gamma,A} \circ \iota^{-1}_{\Delta_1,\Gamma,A} = \iota^{-1}_{\Delta_0,\Gamma,A} \circ \iota_{\Delta_0,\Gamma,A} \circ \delta^* \circ \iota^{-1}_{\Delta_1,\Gamma,A}$$
$$\iota^{-1}_{\Delta_0,\Gamma,A} \circ \textstyle\sum_{\delta^*} \delta^* = \delta^* \circ \iota^{-1}_{\Delta_1,\Gamma,A}$$
□

Exercise 2.13. *For categorically-minded readers: argue that ι is a natural isomorphism in the standard sense, by rephrasing Exercises 2.10 and 2.11 in terms of categories and functors.*

Rather than defining context extension by the collection of rules in Section 2.3 and then characterizing it in terms of ι after the fact, we can actually define it directly as "a context $\Gamma.A$ for which $\mathsf{Sb}(-,\Gamma.A)$ is naturally isomorphic to $\sum_{\gamma\in\mathsf{Sb}(-,\Gamma)} \mathsf{Tm}(-,A[\gamma])$," which unfolds to all of the relevant rules.

In addition to its brevity, the true advantage of such characterizations is that they are less likely to "miss" some important aspect of the definition. Zooming out, this definition states that substitutions into $\Gamma.A$ are dependent pairs of a substitution γ into Γ and a term in $A[\gamma]$, which is exactly the informal description we started with in Section 2.3.

With that in mind, our program for justifying the rules of type theory is as follows:

Slogan 2.4.4. *A connective in type theory is given by (1) a natural type-forming operation and (2) a natural isomorphism relating that type's terms to judgmentally-determined structure.*

We must unfortunately remain vague here about the meaning of "judgmentally-determined structure," but it refers to sets constructed from the sorts $\mathsf{Sb}(\Delta,\Gamma)$, $\mathsf{Ty}(\Gamma)$, and $\mathsf{Tm}(\Gamma,A)$ using natural operations such as dependent products and dependent sums—operations that are implicit in the meaning of inference rules. To make this more precise requires a formal treatment of the algebra of judgments via *logical frameworks*.

In addition, although this slogan will make quick work of the remainder of Section 2.4, we will need to revise it in Sections 2.5 and 2.6.

Π-*types* The rules in Section 2.4.1 precisely capture the existence of an operation

$$\mathbf{\Pi}_\Gamma : (\textstyle\sum_{A\in\mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)) \to \mathsf{Ty}(\Gamma)$$

natural in Γ (that is, one which commutes with substitution) along with the following family of isomorphisms also natural in Γ:

$$\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma,\mathbf{\Pi}(A,B)) \cong \mathsf{Tm}(\Gamma.A,B)$$

The first point expresses the formation rule and $\mathbf{\Pi}(A,B)[\gamma] = \mathbf{\Pi}(A[\gamma],B[\gamma.A])$. We focus on the second point, which characterizes the remaining rules in Section 2.4.1.

The reverse map $\iota^{-1}_{\Gamma,A,B} : \mathsf{Tm}(\Gamma.A,B) \to \mathsf{Tm}(\Gamma,\mathbf{\Pi}(A,B))$ is the introduction rule, which sends terms $\Gamma.A \vdash b : B$ to $\lambda(b)$. The forward map is slightly more involved, but we can guess that it should correspond to elimination. In fact it is *application to a fresh variable*, or a combination of weakening and application—given $\Gamma \vdash f : \mathbf{\Pi}(A,B)$, we weaken to $\Gamma.A \vdash f[\mathbf{p}] : \mathbf{\Pi}(A,B)[\mathbf{p}]$ and then apply to $\mathbf{q}$, obtaining $\Gamma.A \vdash \mathbf{app}(f[\mathbf{p}],\mathbf{q}) : B$.

To complete this natural isomorphism we must check that it is an isomorphism, and that it is natural. We begin with the isomorphism: for all $\vdash \Gamma\ \mathsf{cx}$, $\Gamma \vdash A\ \mathsf{type}$, and $\Gamma.A \vdash B\ \mathsf{type}$,

$$\iota_{\Gamma,A,B}(\iota^{-1}_{\Gamma,A,B}(f)) = f$$
$$\iota^{-1}_{\Gamma,A,B}(\iota_{\Gamma,A,B}(b)) = b$$

Unfolding definitions, we see that this isomorphism boils down essentially to β and η.

$$\begin{aligned}
&\iota^{-1}_{\Gamma,A,B}(\iota_{\Gamma,A,B}(f)) \\
&= \lambda(\mathbf{app}(f[\mathbf{p}],\mathbf{q})) \\
&= f && \text{by the } \eta \text{ rule} \\
&\iota_{\Gamma,A,B}(\iota^{-1}_{\Gamma,A,B}(b)) \\
&= \mathbf{app}(\lambda(b)[\mathbf{p}],\mathbf{q}) \\
&= \mathbf{app}(\lambda(b[\mathbf{p}.A[\mathbf{p}]]),\mathbf{q}) && \lambda(-) \text{ commutes with substitution} \\
&= b[\mathbf{p}.A[\mathbf{p}] \circ \mathbf{id}.\mathbf{q}] && \text{by the } \beta \text{ rule} \\
&= b[\mathbf{p}.\mathbf{q}] && \text{by Exercise 2.14 below} \\
&= b[\mathbf{id}] \\
&= b
\end{aligned}$$

Exercise 2.14. *Using the definition of* $\mathbf{p}.A[\mathbf{p}]$ *from Exercise 2.4, prove the substitution equality needed to complete the equational reasoning above.*

As for the naturality of the isomorphisms ι, as before we must first explain how to relate the types of $\iota_{\Gamma,A,B}$ and $\iota_{\Delta,A[\gamma],B[\gamma.A]}$ given a substitution $\Delta \vdash \gamma : \Gamma$. In this case, the comparison functions are the following:

$$\gamma^* : \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B)) \to \mathsf{Tm}(\Delta, \mathbf{\Pi}(A[\gamma], B[\gamma.A]))$$
$$\gamma.A^* : \mathsf{Tm}(\Gamma.A, B) \to \mathsf{Tm}(\Delta.A[\gamma], B[\gamma.A])$$

Naturality therefore states that "right then down" and "down then right" are equal in the following diagram. (By the reader's argument in Exercise 2.12, naturality of ι

automatically implies the naturality of ι^{-1}.)

$$\begin{tikzcd}
\mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B)) \arrow[r, "\iota_{\Gamma,A,B}"] \arrow[d, "\gamma^*"'] & \mathsf{Tm}(\Gamma.A, B) \arrow[d, "\gamma.A^*"] \\
\mathsf{Tm}(\Delta, \mathbf{\Pi}(A[\gamma], B[\gamma.A])) \arrow[r, "\iota_{\Delta,A[\gamma],B[\gamma.A]}"'] & \mathsf{Tm}(\Delta.A[\gamma], B[\gamma.A])
\end{tikzcd}$$

Fixing $\Gamma \vdash f : \mathbf{\Pi}(A,B)$, we show $\iota_{\Gamma,A,B}(f)[\gamma.A] = \iota_{\Delta,A[\gamma],B[\gamma.A]}(f[\gamma])$ by computing:

$$\begin{aligned}
&\iota_{\Gamma,A,B}(f)[\gamma.A] \\
&\ = \mathbf{app}(f[\mathbf{p}], \mathbf{q})[\gamma.A] \\
&\ = \mathbf{app}(f[\mathbf{p}][\gamma.A], \mathbf{q}[\gamma.A]) && \mathbf{app}(-,-) \text{ commutes with substitution} \\
&\ = \mathbf{app}(f[\mathbf{p} \circ \gamma.A], \mathbf{q}) \\
&\ = \mathbf{app}(f[\gamma \circ \mathbf{p}], \mathbf{q})
\end{aligned}$$

$$\begin{aligned}
&\iota_{\Delta,A[\gamma],B[\gamma.A]}(f[\gamma]) \\
&\ = \mathbf{app}(f[\gamma][\mathbf{p}], \mathbf{q}) \\
&\ = \mathbf{app}(f[\gamma \circ \mathbf{p}], \mathbf{q})
\end{aligned}$$

Thus all of the rules of **Π**-types can be summed up by a natural operation $\mathbf{\Pi}_\Gamma$ (formation and its substitution law) along with a natural isomorphism $\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B)) \cong \mathsf{Tm}(\Gamma.A, B)$ where ι^{-1} and ι are introduction and elimination, the round-trips are β and η, and naturality is the remaining substitution laws.

An alternative eliminator There is a strange asymmetry in the two maps ι and ι^{-1} underlying our natural isomorphism: the latter is literally the introduction rule, but the former combines elimination with weakening and the variable rule. It turns out that there is an equivalent formulation of **Π**-elimination more faithful to our current perspective:

$$\frac{\Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Gamma.A \vdash \lambda^{-1}(f) : B} \Rightarrow$$

Such a presentation replaces the current $\mathbf{app}(-,-)$, β, and η rules with the above rule along with new versions of β and η stating simply that $\lambda(\lambda^{-1}(f)) = f$ and $\lambda^{-1}(\lambda(b)) = b$ respectively. We recover ordinary function application via $\mathbf{app}(f,a) \coloneqq \lambda^{-1}(f)[\mathbf{id}.a]$.

Although in practice our original formulation of function application is much more useful than anti-λ, the latter is more semantically natural. A variant of this argument is discussed by Gratzer et al. [Gra+22], because in the context of *modal type theories* one often encounters elimination forms akin to $\lambda^{-1}(-)$ and it can be far from obvious what the corresponding $\mathbf{app}(-,-)$ operation would be.

Exercise 2.15. *Verify the claim that $\lambda^{-1}(-)$ and its β and η rules do in fact imply our original elimination, β, and η rules.*

2.4.3 *Dependent sums*

We now present dependent pair types, also known as *dependent sums* or Σ*-types*. In a reversal of our discussion of $\mathbf{\Pi}$-types, we will *begin* by defining dependent sums as an internalization of judgmental structure before unfolding this into inference rules.

The Σ type former behaves just like the $\mathbf{\Pi}$ type former: a natural family of types indexed by pairs of a type A and an A-indexed family of types B,

$$\Sigma_\Gamma : (\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)) \longrightarrow \mathsf{Ty}(\Gamma)$$

or in inference rule notation,

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Gamma \vdash \Sigma(A,B)\,\mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Delta \vdash \Sigma(A,B)[\gamma] = \Sigma(A[\gamma], B[\gamma.A])\,\mathsf{type}}$$

(Recall that we write $\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)$ for the indexed coproduct $\coprod_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)$.)

Where Σ-types and $\mathbf{\Pi}$-types differ is in their elements. Whereas $\Gamma \vdash \mathbf{\Pi}(A,B)$ type internalizes terms with a free variable $\Gamma.A \vdash b : B$, the type $\Gamma \vdash \Sigma(A,B)$ type internalizes pairs of terms $\Gamma \vdash a : A$ and $\Gamma \vdash b : B[\mathbf{id}.a]$, naturally in Γ:

$$\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, \Sigma(A,B)) \cong \textstyle\sum_{a \in \mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])$$

Remarkably, the above line completes our definition of dependent sum types, but in the interest of the reader we will proceed to unfold this natural isomorphism into inference rules in three stages. First, we will unfold the maps $\iota_{\Gamma,A,B}$ and $\iota^{-1}_{\Gamma,A,B}$ into three term formers; second, we will unfold the two round-trip equations into a pair of equational rules; and finally, we will unfold the naturality condition into three more equational rules.

Remark 2.4.5. There is an unfortunate terminological collision between simple types and dependent types: although $\mathbf{\Pi}$-types seem to generalize *simple functions*, they are called *dependent products*, and although Σ-types seem to generalize *simple products* because their elements are pairs, they are called *dependent sums*.

The reason is twofold: first, the elements of indexed coproducts (known to programmers as "tagged unions") are actually pairs ("pairs of a tag bit with data"), whereas the elements of indexed products ("n-ary pairs") are actually functions (sending n to the n-th projection). Secondly, *both concepts* generalize simple finite products: the product $B_1 \times B_2$ is both an indexed product $\prod_{a \in \{1,2\}} B_a$ and an indexed coproduct of a constant family $\sum_{\_ \in B_1} B_2$. $\diamond$

To unpack the natural isomorphism, we note first that the forward direction $\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, \Sigma(A,B)) \to \sum_{a\in\mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])$ sends terms $\Gamma \vdash p : \Sigma(A,B)$ to (meta-)pairs of terms, so we can unfold this map into a pair of term formers with the same premises:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{fst}(p) : A}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{snd}(p) : B[\mathbf{id}.\mathbf{fst}(p)]}$$

The map $\iota^{-1}_{\Gamma,A,B} : \sum_{a\in\mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a]) \to \mathsf{Tm}(\Gamma, \Sigma(A,B))$ sends a pair of terms to a single term of type $\Sigma(A,B)$, so we unfold it into one term former with two term premises:

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{pair}(a,b) : \Sigma(A,B)}$$

Unlike in our judgmental analysis of dependent products, the standard introduction and elimination forms of dependent sums correspond exactly to the maps ι^{-1} and ι, so the two round-trip equations are exactly the standard β and η principles:

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{fst}(\mathbf{pair}(a,b)) = a : A \qquad \Gamma \vdash \mathbf{snd}(\mathbf{pair}(a,b)) = b : B[\mathbf{id}.a]}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash p = \mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p)) : \Sigma(A,B)}$$

It remains to unpack the naturality of ι, which as we have seen previously, encodes the fact that the term formers commute with substitution. The reader may be surprised to learn, however, that the substitution rule for $\mathbf{pair}(-,-)$ actually implies the substitution rules for $\mathbf{fst}(-)$ and $\mathbf{snd}(-)$ in the presence of β and η. (Categorically, this is the fact that naturality of ι^{-1} implies naturality of ι, as we saw in Exercise 2.12.) Given the rule

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Delta \vdash \mathbf{pair}(a,b)[\gamma] = \mathbf{pair}(a[\gamma], b[\gamma]) : \Sigma(A,B)[\gamma]}$$

fix a substitution $\Delta \vdash \gamma : \Gamma$ and a term $\Gamma \vdash p : \Sigma(A,B)$. Then

$$\begin{aligned}
&\mathbf{fst}(p)[\gamma] \\
&= \mathbf{fst}(\mathbf{pair}(\mathbf{fst}(p)[\gamma], \mathbf{snd}(p)[\gamma])) && \text{by the } \beta \text{ rule} \\
&= \mathbf{fst}(\mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p))[\gamma]) && \text{by the above rule} \\
&= \mathbf{fst}(p[\gamma]) && \text{by the } \eta \text{ rule}
\end{aligned}$$

and the calculation for $\mathbf{snd}(-)$ is identical. Nevertheless it is typical to include substitution rules for all three term formers: there is nothing wrong with equating terms that are already equal, and even in type theory, discretion can be the better part of valor.

Exercise 2.16. *Check that the substitution rule for* **pair** *above is meta-well-typed, in particular the second component* $b[\gamma]$. *(Hint: use Exercise 2.3.)*

Exercise 2.17. *Show that the substitution rule for* $\lambda^{-1}(-)$ *follows from the substitution rule for* $\lambda(-)$ *and the equations* $\lambda(\lambda^{-1}(f)) = f$ *and* $\lambda^{-1}(\lambda(b)) = b$.

2.4.4 Extensional equality

We now turn to the simplest form of propositional equality, known as *extensional equality* or **Eq**-*types*. As their name suggests, **Eq**-types internalize the term equality judgment. They are defined as follows, naturally in Γ:

$$\mathbf{Eq}_\Gamma : (\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Tm}(\Gamma, A) \times \mathsf{Tm}(\Gamma, A)) \to \mathsf{Ty}(\Gamma)$$
$$\iota_{\Gamma,A,a,b} : \mathsf{Tm}(\Gamma, \mathbf{Eq}(A, a, b)) \cong \{\star \mid a = b\}$$

In other words, $\mathbf{Eq}(A, a, b)$ is a type when $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$, and has a unique inhabitant exactly when the judgment $\Gamma \vdash a = b : A$ holds (otherwise it is empty). The inference rules for extensional equality are as follows:

$$\frac{\Gamma \vdash a, b : A}{\Gamma \vdash \mathbf{Eq}(A, a, b)\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a, b : A}{\Delta \vdash \mathbf{Eq}(A, a, b)[\gamma] = \mathbf{Eq}(A[\gamma], a[\gamma], b[\gamma])\ \mathsf{type}}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl} : \mathbf{Eq}(A, a, a)} \qquad \frac{\Gamma \vdash a, b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash a = b : A}$$

$$\frac{\Gamma \vdash a, b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash p = \mathbf{refl} : \mathbf{Eq}(A, a, b)}$$

The penultimate rule is known as *equality reflection*, and it is somewhat unusual because it concludes an arbitrary term equality judgment from the existence of a term. This rule is quite strong in light of the facts that (1) judgmentally equal terms can be silently exchanged at any location in any judgment, (2) the equality proof $\Gamma \vdash p : \mathbf{Eq}(A, a, b)$ is not recorded in those exchanges, and (3) p could even be a variable, e.g., in context $\Gamma.\mathbf{Eq}(A, a, b)$.

Type theories with an extensional equality type are called *extensional.* The consequences of equality reflection will be the primary motivation behind the latter half of these lecture notes, but for now we simply note that these rules are a very natural axiomatization of an equality type as the internalization of equality.

Exercise 2.18. *Explain how these inference rules correspond to our* $\mathbf{Eq}_\Gamma$ *and* $\iota_{\Gamma,A,a,b}$ *definition.*

Exercise 2.19. *Where are the substitution rules for term formers? (Hint: there are two equivalent answers, in terms of either the natural isomorphism or the inference rules.)*

2.4.5 *The unit type*

We conclude our tour of the best-behaved connectives of type theory with the simplest connective of all: the unit type.

$$\mathbf{Unit}_\Gamma \in \mathsf{Ty}(\Gamma)$$
$$\iota_\Gamma : \mathsf{Tm}(\Gamma, \mathbf{Unit}) \cong \{\star\}$$

This unfolds to the following rules:

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Unit}\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Unit}[\gamma] = \mathbf{Unit}\ \mathsf{type}}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{tt} : \mathbf{Unit}} \qquad \frac{\Gamma \vdash a : \mathbf{Unit}}{\Gamma \vdash a = \mathbf{tt} : \mathbf{Unit}}$$

Exercise 2.20. *Where is the elimination principle? Where are the substitution rules for term formers? (Hint: what would these say in terms of the natural isomorphism?)*

2.5 *Inductive types:* Void, Bool, Nat

We now turn our attention to *inductive types*, data types with induction principles. Unlike the type formers in Section 2.4, which are typically "hard coded" into type theories,[8] inductive types are usually specified by users as extensions to the theory via inductive schemas [Dyb94; CP90] (essentially, data type declarations), or in theoretical contexts, encoded as well-founded trees known as **W**-types [ML82; ML84]. These schemas can be extended *ad infinitum* to account for increasingly complex forms of inductive definition, including indexed induction [Dyb94], mutual induction, induction-recursion [Dyb00], induction-induction [NFS12], quotient induction-induction [KKA19], and so forth.

For simplicity we restrict our attention to three examples—the empty type, booleans, and natural numbers—that illustrate the basic issues that arise when specifying inductive types in type theory. Unfortunately, we will immediately need to refine Slogan 2.4.4.

[8] This is an oversimplification: in practice, Σ and **Unit** are usually obtained as special cases of *dependent record types* [Pol02], n-ary Σ-types with named projections.

2.5.1 *The empty type*

We begin with the empty type **Void**, a "type with no elements." Logically, this type corresponds to the false proposition, so there should be no way to construct an element of **Void** (a proof of false) except by deriving a contradiction from local hypotheses. The type former is straightforward: naturally in Γ, a constant $\mathbf{Void}_\Gamma \in \mathsf{Ty}(\Gamma)$, or

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Void}\ \mathsf{type}} \qquad\qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Void}[\gamma] = \mathbf{Void}\ \mathsf{type}}$$

As for the elements of **Void**, an obvious guess is to say that the elements of the empty type at each context are the empty set, i.e., naturally in Γ,

$$\iota_\Gamma : \mathsf{Tm}(\Gamma, \mathbf{Void}) \cong \emptyset \tag{!?}$$

This cannot be right, however, because **Void** *does* have elements in some contexts—the variable rule alone forces $\mathbf{q} \in \mathsf{Tm}(\Gamma.\mathbf{Void}, \mathbf{Void})$, and other type formers can populate **Void** even further, e.g., $\mathbf{app}(\mathbf{q}, \mathbf{tt}) \in \mathsf{Tm}(\Gamma.\mathbf{\Pi}(\mathbf{Unit}, \mathbf{Void}), \mathbf{Void})$.

Interlude: mapping in, mapping out To see how to proceed, let us take a brief sojourn into set theory. There are several ways to define the product $A \times B$ of two sets, for example by constructing it as the set of ordered pairs $\{(a, b) \mid a \in A \wedge b \in B\}$ or even more explicitly as the set $\{\{\{a\}, \{a, b\}\} \mid a \in A \wedge b \in B\}$. However, in addition to these explicit constructions, it is also possible to *characterize* the set $A \times B$ up to isomorphism, as the set such that every function $X \longrightarrow A \times B$ is determined by a pair of functions $X \longrightarrow A$ and $X \longrightarrow B$ and vice versa.

Similarly, we can characterize one-element sets $\mathbf{1}$ as those sets for which there is exactly one function $X \longrightarrow \mathbf{1}$ for all sets X. In fact, both of these characterizations are set-theoretical analogues of Slogan 2.4.4, where X plays the role of the context Γ.

After some thought, we realize that the analogous characterization of the zero-element (empty) set $\mathbf{0}$ is significantly more awkward: there is exactly one function $X \longrightarrow \mathbf{0}$ when X is itself empty, and no functions $X \longrightarrow \mathbf{0}$ when X is non-empty. As it turns out, in this case it is more elegant to consider the functions *out* of $\mathbf{0}$ rather than the functions *into* it: a zero-element set $\mathbf{0}$ has exactly one function $\mathbf{0} \longrightarrow X$ for all sets X.

Exercise 2.21. *Suppose that Z is a set such that for all sets X there is exactly one function $Z \longrightarrow X$. Show that Z is isomorphic to the empty set.*

Void revisited Recall from Section 2.3 that terms correspond to "dependent functions from Γ to A." In Section 2.4 we considered only type formers T that are easily characterized in terms of the maps *into* that type former from an arbitrary context Γ: in each case we defined maps/terms $\mathsf{Tm}(\Gamma, T)$ as naturally isomorphic to the data of T's introduction rule.

To characterize the maps *out of* **Void** into an arbitrary type A, we cannot leave the context fully unconstrained; instead, we must characterize the maps/terms $\mathsf{Tm}(\Gamma.\mathbf{Void}, A)$ for all $\vdash \Gamma\ \mathsf{cx}$ and $\Gamma.\mathbf{Void} \vdash A\ \mathsf{type}$, recalling that—by the rules for $\mathbf{\Pi}$-types—these are equivalently the dependent functions out of **Void** in context Γ, i.e., $\Gamma \vdash f : \mathbf{\Pi}(\mathbf{Void}, A)$.

Advanced Remark 2.5.1. Writing $\mathcal{C}$ for the category of contexts and substitutions, terms $\mathsf{Tm}(\Gamma, A)$ are indeed "dependent morphisms" from Γ to A; more precisely, by Exercise 2.2, they are ordinary morphisms $\Gamma \to \Gamma.A$ in the slice category $\mathcal{C}/\Gamma$. Thus, for *right adjoint* type operations G—those in Section 2.4—it is easy to describe $\mathsf{Tm}(\Gamma, G(A))$ directly.

For *left adjoint* type operations F, the situation is more fraught. Type theory is fundamentally "right-biased" because its judgments concern maps from arbitrary contexts *into* fixed types, but not vice versa. Thus to discuss dependent morphisms $F(X) \to A$ we must speak about elements of $\mathsf{Tm}(\Gamma.F(X), A)$, quantifying not only over the ambient context/slice Γ but also the type A into which we are mapping.

Confusingly, we encountered no issues defining Σ-types, despite dependent sum being the left adjoint to pullback. This is because Σ is also the right adjoint to the functor $\mathcal{C} \to \mathcal{C}^{\to}$ sending $A \mapsto \mathbf{id}_A$, and it is the latter perspective that we axiomatize. The left adjoint axiomatization makes an appearance in some systems—particularly in the context of programming languages with existential types—phrased as **let** $(a, b) = p$ in x. ⋄

Putting all these ideas together, we will define **Void** as the type for which, naturally in Γ, there is exactly one dependent function from **Void** to A for any dependent type A:

$$\rho_{\Gamma,A} : \mathsf{Tm}(\Gamma.\mathbf{Void}, A) \cong \{\star\}$$

To sum up the difference between the incorrect definition $\mathsf{Tm}(\Gamma, \mathbf{Void}) \cong \emptyset$ and the correct one above, the former states that $\mathsf{Tm}(\Gamma, \mathbf{Void})$ is the smallest set (in the sense of mapping into all other sets), whereas the latter states that in any context, **Void** is the smallest *type*. More poetically, at the level of judgments we can see that **Void** is not always empty, but at the level of types, every type "believes" that **Void** is empty.

Unwinding $\rho_{\Gamma,A}$ into inference rules, we obtain:

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathsf{type}}{\Gamma.\mathbf{Void} \vdash \mathbf{absurd}' : A}\ ✎ \qquad\qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Void} \vdash a : A}{\Gamma.\mathbf{Void} \vdash \mathbf{absurd}' = a : A}\ ✎$$

We have marked these rules with ✎ to indicate that they are provisional; in practice, as we previously discussed for $\lambda^{-1}(-)$, it is awkward to use rules whose conclusions constrain the shape of their context. But just as with $\mathbf{app}(-,-)$, it is more standard to present an equivalent axiomatization $\mathbf{absurd}(b) \coloneqq \mathbf{absurd}'[\mathbf{id}.b]$ that "builds in a cut":

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\,\mathsf{type}}{\Gamma \vdash \mathbf{absurd}(b) : A[\mathbf{id}.b]} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\,\mathsf{type}}{\Delta \vdash \mathbf{absurd}(b)[\gamma] = \mathbf{absurd}(b[\gamma]) : A[\gamma.b[\gamma]]}$$

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash a : A}{\Gamma \vdash \mathbf{absurd}(b) = a[\mathbf{id}.b] : A[\mathbf{id}.b]}\ ✎$$

The term $\mathbf{absurd}(-)$ is known as the *induction principle* for **Void**, in the sense that it allows users to prove a theorem for all terms of type **Void** by proving that it holds for each constructor of **Void**, of which there are none.

In light of our definition of **Void**, we update Slogan 2.4.4 as follows:

Slogan 2.5.2. *A connective in type theory is given by (1) a natural type-forming operation* Υ *and (2) one of the following:*

2.1. *a natural isomorphism relating* $\mathsf{Tm}(\Gamma, \Upsilon)$ *to judgmentally-determined structure, or*

2.2. *for all* $\Gamma.\Upsilon \vdash A\,\mathsf{type}$*, a natural isomorphism relating* $\mathsf{Tm}(\Gamma.\Upsilon, A)$ *to judgmentally-determined structure.*

The final rule for $\mathbf{absurd}(-)$, the η principle, implies a very strong equality principle for terms in an inconsistent context (Exercise 2.25) which we derive in the following sequence of exercises. For this reason, and because this rule is derivable in the presence of extensional equality (Section 2.5.4), we consider it provisional ✎ for the time being.

Exercise 2.22. *Show that if* $\Gamma \vdash b_0, b_1 : \mathbf{Void}$ *then* $\Gamma \vdash b_0 = b_1 : \mathbf{Void}$.

Exercise 2.23. *Fixing* $\Delta \vdash \gamma : \Gamma$*, prove that there is at most one substitution* $\Delta \vdash \bar{\gamma} : \Gamma.\mathbf{Void}$ *satisfying* $\mathbf{p} \circ \bar{\gamma} = \gamma$.

Exercise 2.24. *Let* $\Gamma.\mathbf{Void} \vdash A\,\mathsf{type}$ *and* $\Gamma \vdash a : A[\mathbf{id}.b]$*. Show that* $\Gamma.\mathbf{Void} \vdash A[\mathbf{id}.b \circ \mathbf{p}] = A\,\mathsf{type}$*, and therefore that* $\Gamma.\mathbf{Void} \vdash a[\mathbf{p}] : A$.

Exercise 2.25. *Derive the following rule, using the previous exercise as well as the* η *rule.*

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a : A[\mathbf{id}.b]}{\Gamma \vdash a = \mathbf{absurd}(b) : A[\mathbf{id}.b]}\ \Rightarrow$$

Exercise 2.26. *We have included the rule* $\Delta \vdash \mathbf{absurd}(b)[\gamma] = \mathbf{absurd}(b[\gamma]) : A[\gamma.b[\gamma]]$ *but it is in fact derivable using the* η *rule. Prove this.*

2.5.2 Booleans

We turn now to the booleans **Bool**, a "type with two elements." Once again the type former is straightforward: $\mathbf{Bool}_\Gamma \in \mathsf{Ty}(\Gamma)$ naturally in Γ, or

$$\frac{}{\Gamma \vdash \mathbf{Bool}\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Bool}[\gamma] = \mathbf{Bool}\ \mathsf{type}}$$

It is also clear that we want two constructors of **Bool**, **true** and **false**, natural in Γ:

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{true} = \mathbf{true}[\gamma] : \mathbf{Bool}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{false} = \mathbf{false}[\gamma] : \mathbf{Bool}}$$

Keeping Slogan 2.5.2 in mind, there are two possible ways for us to complete our axiomatization of **Bool**. As with **Void** it is tempting but incorrect to define $\iota : \mathsf{Tm}(\Gamma, \mathbf{Bool}) \cong \{\star, \star'\}$; although the natural transformation ι^{-1} is equivalent to our rules for **true** and **false**, ι does not account for variables of type **Bool** or other indeterminate booleans that arise in non-empty contexts.[9] Thus we must instead characterize maps *out of* **Bool** by giving a family of sets naturally isomorphic to $\mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$.

So, what should terms $\Gamma.\mathbf{Bool} \vdash a : A$ be? By substitution, such a term clearly determines a pair of terms $\Gamma \vdash a[\mathbf{id}.\mathbf{true}] : A[\mathbf{id}.\mathbf{true}]$ and $\Gamma \vdash a[\mathbf{id}.\mathbf{false}] : A[\mathbf{id}.\mathbf{false}]$. Conversely, if **true** and **false** are the "only" booleans, then such a pair of terms should uniquely determine elements of $\mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$ in the sense that to map out of **Bool**, it suffices to explain what to do on **true** and on **false**.

To formalize this idea, let us write $((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*)$ for the function which sends $a \in \mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$ to the pair $(a[\mathbf{id}.\mathbf{true}], a[\mathbf{id}.\mathbf{false}])$. We complete our specification of **Bool** by asking for this map to be a natural isomorphism; thus, naturally in Γ, we have:

$$\mathbf{Bool}_\Gamma \in \mathsf{Ty}(\Gamma)$$
$$\mathbf{true}_\Gamma, \mathbf{false}_\Gamma \in \mathsf{Tm}(\Gamma, \mathbf{Bool})$$
$$((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*) : \mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \cong \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}]) \times \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}])$$

This definition is remarkable in several ways. For the first time we are asking not only for the existence of some natural isomorphism, but for a *particular map* to be a natural isomorphism; and because this map is defined in terms of **true** and **false**, these must be asserted prior to the natural isomorphism itself. We update our slogan accordingly:

[9] Even if variables $x : \mathbf{Bool}$ stand for one of **true** or **false**, x itself must be an indeterminate boolean equal to neither constructor; otherwise the identity $\lambda x.x : \mathbf{Bool} \to \mathbf{Bool}$ would be a constant function.

Slogan 2.5.3. *A connective in type theory is given by (1) a natural type-forming operation* Υ *and (2) one of the following:*

2.1. *a natural isomorphism relating* $\mathsf{Tm}(\Gamma, \Upsilon)$ *to judgmentally-determined structure, or*

2.2. *a collection of natural term constructors for* Υ *which, for all* $\Gamma.\Upsilon \vdash A\ \mathsf{type}$*, determine a natural isomorphism relating* $\mathsf{Tm}(\Gamma.\Upsilon, A)$ *to judgmentally-determined structure.*

In the case of **Void** we simply had no term constructors to specify, and because there is at most one (natural) isomorphism between anything and $\{\star\}$, it was unnecessary for us to specify the underlying map. In general, however, we emphasize that it is essential to specify the map; this is what ensures that when we define a function "by cases" on **true** and **false**, applying it to **true** or **false** recovers the specified case and not something else. On the other hand, because we have specified the underlying map, it being an isomorphism is a *property* rather than additional structure: there is at most one possible inverse.

Zooming out, however, our definition of **Bool** has a similar effect to our definition of **Void** from Section 2.5.1: $\mathsf{Tm}(\Gamma, \mathbf{Bool})$ is *not* the set $\{\mathbf{true}, \mathbf{false}\}$ at the level of judgments, but every type "believes" that it is. This is the role of type-theoretic induction principles.

Advanced Remark 2.5.4. From the categorical perspective, option 2.2 in Slogan 2.5.3 asserts that the inclusion map of Υ's constructors into Υ's terms is *left orthogonal* to all types. Maps which are left orthogonal to a class of objects and whose codomain belongs to that class are known as *fibrant replacements*; in this sense, we have defined $\mathsf{Tm}(-, \mathbf{Void})$ and $\mathsf{Tm}(-, \mathbf{Bool})$ as fibrant replacements of the constantly zero- and two-element presheaves. This perspective is crucial to early work in homotopy type theory [AW09] and the formulation of the intensional identity type in natural models [Awo18]. $\diamond$

It remains to unfold our natural isomorphism into inference rules. We do not need any additional rules for the forward map, which is substitution by $\mathbf{id}.\mathbf{true}$ and $\mathbf{id}.\mathbf{false}$. As the reader may have already guessed, the backward map is essentially[10] dependent **if**:

$$\frac{\Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}] \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}(a_t, a_f, b) : A[\mathbf{id}.b]}$$

$$\frac{\begin{array}{c}\Delta \vdash \gamma : \Gamma \\ \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}] \qquad \Gamma \vdash b : \mathbf{Bool}\end{array}}{\Delta \vdash \mathbf{if}(a_t, a_f, b)[\gamma] = \mathbf{if}(a_t[\gamma], a_f[\gamma], b[\gamma]) : A[\gamma.b[\gamma]]}$$

[10] The inverse directly lands in $\Gamma.\mathbf{Bool}$ and not Γ, but as with **absurd′** (Section 2.5.1) we adopt a more standard presentation in which all conclusions have a generic context; see Exercise 2.27.

The fact that **if** is an inverse to $((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*)$ expresses the β and η laws:

$$\frac{\Gamma.\mathbf{Bool} \vdash A \;\mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}(a_t, a_f, \mathbf{true}) = a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash \mathbf{if}(a_t, a_f, \mathbf{false}) = a_f : A[\mathbf{id}.\mathbf{false}]}$$

$$\frac{\Gamma.\mathbf{Bool} \vdash A \;\mathsf{type} \qquad \Gamma.\mathbf{Bool} \vdash a : A \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}(a[\mathbf{id}.\mathbf{true}], a[\mathbf{id}.\mathbf{false}], b) = a[\mathbf{id}.b] : A[\mathbf{id}.b]}$$

The β laws—the first two equations—are perhaps more familiar than the η law, which effectively asserts that any two terms dependent on **Bool** are equal if (and only if) they are equal on **true** and **false**. (The η rule is sometimes decomposed into a "local expansion" and a collection of "commuting conversions.") Although semantically justified, it is typical to omit judgmental η laws for all inductive types because they are not syntax-directed and thus challenging to implement, and because they are derivable in the presence of extensional equality (Section 2.5.4).

Exercise 2.27. *Give rules axiomatizing the boolean analogue of* $\mathbf{absurd}'$, *and prove that these rules are interderivable with our rules for* $\mathbf{if}(a_t, a_f, b)$.

2.5.3 *Natural numbers*

Our final example of an inductive type is the type of natural numbers **Nat**, the "least type closed under $\mathbf{zero} : \mathbf{Nat}$ and $\mathbf{suc}(-) : \mathbf{Nat} \to \mathbf{Nat}$." The natural numbers more or less fit the same pattern as **Void** and **Bool**, but the recursive nature of $\mathbf{suc}(-)$ complicates the situation significantly. The formation and introduction rules remain straightforward:

$$\frac{}{\Gamma \vdash \mathbf{Nat} \;\mathsf{type}} \qquad \frac{}{\Gamma \vdash \mathbf{zero} : \mathbf{Nat}} \qquad \frac{\Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{suc}(n) : \mathbf{Nat}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Nat}[\gamma] = \mathbf{Nat} \;\mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{zero}[\gamma] = \mathbf{zero} : \mathbf{Nat}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{suc}(n)[\gamma] = \mathbf{suc}(n[\gamma]) : \mathbf{Nat}}$$

Following the pattern we established with **Bool**, we might ask for maps out of **Nat** to be determined by their behavior on **zero** and $\mathbf{suc}(-)$, i.e., for the two substitutions

$$(\mathbf{id}.\mathbf{zero})^* : \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \to \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{zero}])$$
$$(\mathbf{p}.\mathbf{suc}(\mathbf{q}))^* : \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \to \mathsf{Tm}(\Gamma.\mathbf{Nat}, A[\mathbf{p}.\mathbf{suc}(\mathbf{q})])$$

to determine, for every $\Gamma.\mathbf{Nat} \vdash A$ type, a natural isomorphism

$$((\mathbf{id}.\mathbf{zero})^*, (\mathbf{p}.\mathbf{suc}(\mathbf{q}))^*) :\\ \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \cong \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{zero}]) \times \mathsf{Tm}(\Gamma.\mathbf{Nat}, A[\mathbf{p}.\mathbf{suc}(\mathbf{q})]) \tag{!?}$$

This turns out to not be the correct definition, but first, note that the first substitution moves us from $\Gamma.\mathbf{Nat}$ to Γ (analogously to **Bool**) whereas the second substitution moves us from $\Gamma.\mathbf{Nat}$ also to $\Gamma.\mathbf{Nat}$; this is because the $\mathbf{suc}(-)$ constructor has type "$\mathbf{Nat} \to \mathbf{Nat}$," so the condition of "being determined by one's behavior on $\mathbf{suc}(n) : \mathbf{Nat}$" is properly stated relative to a variable $n : \mathbf{Nat}$. Put more simply, if the argument of $\mathbf{suc}(-)$ was of type X rather than **Nat**, then the latter substitution would be $\Gamma.X \vdash \mathbf{p}.\mathbf{suc}(\mathbf{q}) : \Gamma.\mathbf{Nat}$.

But given that $\mathbf{suc}(-)$ *is* recursive—taking **Nat** to **Nat**—we now for the first time are defining a judgment by a natural isomorphism whose *right-hand* side *also* has the very same judgment we are trying to define, namely $\mathsf{Tm}(\Gamma.\mathbf{Nat}, \dots)$, i.e., terms in context $\Gamma.\mathbf{Nat}$. This natural isomorphism is therefore not so much a *definition* of its left-hand side as it is an *equation* that the left-hand side must satisfy—in principle, this equation may have many different solutions for $\mathsf{Tm}(\Gamma.\mathbf{Nat}, A)$, or no solutions at all.

Interlude: initial algebras This equation asserts in essence that the natural numbers are a set N satisfying the isomorphism $N \cong \{\star\} + N$,[11] where the reverse map equips N with a choice of "implementations" of $\mathbf{zero} \in N$ and $\mathbf{suc}(-) : N \to N$. The set of natural numbers $\mathbb{N}$ with $\mathbf{zero} \coloneqq 0$ and $\mathbf{suc}(n) \coloneqq n + 1$ are a solution, but there are infinitely many *other* solutions as well, such as $\mathbb{N} + \{\infty\}$ with $\mathbf{zero} \coloneqq 0$, $\mathbf{suc}(n) \coloneqq n + 1$, and $\mathbf{suc}(\infty) \coloneqq \infty$.

Nevertheless one might imagine that $(\mathbb{N}, 0, - + 1)$ is a distinguished solution in some way, and indeed it is the "least" set N with a point $z \in N$ and endofunction $s : N \to N$—here we are dropping the requirement of (z, s) being an isomorphism—in the sense that for any (N, z, s) there is a unique function $f : \mathbb{N} \to N$ with $f(0) = z$ and $f(n+1) = s(f(n))$. Such triples (N, z, s) are known as *algebras* for the signature $N \mapsto 1 + N$, structure-preserving functions between algebras are known as *algebra homomorphisms*, and algebras with the above minimality property are *initial algebras*.

The above definitions extend straightforwardly to dependent algebras and homomorphisms: given an ordinary algebra (N, z, s), a *displayed algebra over* (N, z, s) is a triple of an N-indexed family of sets $\{\tilde{N}_n\}_{n \in N}$, an element $\tilde{z} \in \tilde{N}_z$, and a function $\tilde{s} : (n : N) \to \tilde{N}_n \to \tilde{N}_{s(n)}$ [KKA19]. Given any displayed algebra $(\tilde{N}, \tilde{z}, \tilde{s})$ over the natural number algebra $(\mathbb{N}, 0, - + 1)$, there is once again a unique function $f : (n : \mathbb{N}) \to \tilde{N}_n$ with $f(0) = \tilde{z}$ and $f(n + 1) = \tilde{s}(n, f(n))$. The reader is likely familiar with the special case of displayed algebras over $\mathbb{N}$ valued in *propositions* rather than sets:

$$\forall P : \mathbb{N} \to \mathbf{Prop}.\ P(0) \implies (\forall n.P(n) \implies P(n+1)) \implies \forall n.P(n)$$

[11] Why? In algebraic notation and ignoring dependency, the equation states that $A^{\Gamma \times N} \cong A^{\Gamma} \times A^{\Gamma \times N}$, which simplifies to $(\Gamma \times N) \cong \Gamma + (\Gamma \times N)$ and thus $N \cong 1 + N$.

Advanced Remark 2.5.5. The data of a displayed algebra over (N, z, s) is equivalent to the data of an algebra homomorphism into (N, z, s), where the forward direction of this equivalence sends the family $\{\tilde{N}_n\}_{n\in N}$ to the first projection $(\sum_{n\in N} \tilde{N}_n) \to N$. A displayed algebra over the natural number algebra is thus a homomorphism $\tilde{N} \to \mathbb{N}$; the initiality of $\mathbb{N}$ implies this map has a unique section homomorphism, which unfolds to the dependent universal property stated above. ⋄

Natural numbers revisited Coming back to our specification of **Nat**, our formation and introduction rules axiomatize an algebra $(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(-))$ for the signature $N \mapsto 1 + N$, but our proposed **Bool**-style natural isomorphism does not imply that this algebra is initial. The solution is to simply axiomatize that any displayed algebra over $(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(-))$ admits a unique displayed algebra homomorphism from $(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(-))$.

Unwinding definitions, we ask that naturally in Γ, and for any $A \in \mathsf{Ty}(\Gamma.\mathbf{Nat})$, $a_z \in \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{zero}])$, and $a_s \in \mathsf{Tm}(\Gamma.\mathbf{Nat}.A, A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])])$, we have an isomorphism:

$$\rho_{\Gamma,A,a_z,a_s} : \{a \in \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \mid a_z = a[\mathbf{id}.\mathbf{zero}] \wedge a_s[\mathbf{p}.\mathbf{q}.a] = a[\mathbf{p}.\mathbf{suc}(\mathbf{q})]\} \cong \{\star\}$$

The type of a_s is easier to understand with named variables: it is a term of type $A(\mathbf{suc}(n))$ in context $\Gamma, n : \mathbf{Nat}, a : A(n)$.

Remark 2.5.6. This is the third time we have defined a connective in terms of a natural isomorphism with $\{\star\}$. In Section 2.4.5, we used such an isomorphism to assert that **Unit** has a unique element in every context; in Section 2.5.1, we asserted dually that every dependent type over **Void** admits a unique dependent function from **Void**. The present definition is analogous to the latter, but restricted to algebras: every displayed algebra over **Nat** admits a unique displayed algebra homomorphism from **Nat**. ⋄

Advanced Remark 2.5.7. In light of Remark 2.5.4 and Remark 2.5.6, we have defined **Nat** as the fibrant replacement of the initial object in the category of $(1 + -)$-algebras. ⋄

In rule form, the reverse direction of the natural isomorphism states that any displayed algebra (A, a_z, a_s) over **Nat** gives rise to a map out of **Nat**,

$$\frac{\Gamma \vdash n : \mathbf{Nat} \qquad \Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Gamma \vdash \mathbf{rec}(a_z, a_s, n) : A[\mathbf{id}.n]}$$

which commutes with substitution,

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash n : \mathbf{Nat} \qquad \Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Delta \vdash \mathbf{rec}(a_z, a_s, n)[\gamma] = \mathbf{rec}(a_z[\gamma], a_s[\gamma.\mathbf{Nat}.A], n[\gamma]) : A[\gamma.n[\gamma]]}$$

and is a displayed algebra homomorphism, i.e., sends **zero** to a_z and $\mathbf{suc}(n)$ to $a_s(n, \mathbf{rec}(n))$:

$$\frac{\Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{zero}) = a_z : A[\mathbf{id}.\mathbf{zero}]}$$

$$\frac{\begin{array}{c}\Gamma \vdash n : \mathbf{Nat} \\ \Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]\end{array}}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{suc}(n)) = a_s[\mathbf{id}.n.\mathbf{rec}(a_z, a_s, n)] : A[\mathbf{id}.\mathbf{suc}(n)]}$$

Finally, the η rule of **Nat**, which is again typically omitted, expresses that there is exactly one displayed algebra homomorphism from **Nat** to (A, a_z, a_s): if $\Gamma.\mathbf{Nat} \vdash a : A$ is a term that sends **zero** to a_z and $\mathbf{suc}(n)$ to $a_s(n, a[\mathbf{id}.n])$, then it is equal to $\mathbf{rec}(a_z, a_s, \mathbf{q})$.

$$\frac{\begin{array}{c}\Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma.\mathbf{Nat} \vdash a : A \qquad \Gamma \vdash n : \mathbf{Nat} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma \vdash a_z = a[\mathbf{id}.\mathbf{zero}] : A[\mathbf{id}.\mathbf{zero}] \\ \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \qquad \Gamma.\mathbf{Nat} \vdash a_s[\mathbf{p}.\mathbf{q}.a] = a[\mathbf{p}.\mathbf{suc}(\mathbf{q})] : A[\mathbf{p}.\mathbf{suc}(\mathbf{q})]\end{array}}{\Gamma \vdash \mathbf{rec}(a_z, a_s, n) = a[\mathbf{id}.n] : A[\mathbf{id}.n]}$$ ✎

Exercise 2.28. *Rewrite the first* **rec** *rule using named variables instead of* **p** *and* **q**, *and convince yourself that it expresses a form of natural number induction.*

Exercise 2.29. *Define addition for* **Nat** *in terms of* **rec**. *We strongly recommend solving* Exercise 2.28 *prior to this exercise in order to use standard named syntax.*

Inductive types are initial algebras Our definition of **Nat** is more similar to our definitions of **Void** and **Bool** than it may first appear. In fact, all three types are initial algebras for different signatures, although the absence of recursive constructors in **Void** and **Bool** allowed us to sidestep this machinery. The empty type **Void** is the initial algebra for the signature $X \mapsto \mathbf{0}$: a (displayed) $\mathbf{0}$-algebra is just a (dependent) type with no additional data, so initiality asserts that any $\Gamma.\mathbf{Void} \vdash A\,\mathsf{type}$ admits a unique displayed algebra homomorphism—a dependent function with no additional conditions—from **Void**.

Likewise, $(\mathbf{Bool}, \mathbf{true}, \mathbf{false})$ is the initial algebra for the signature $X \mapsto \mathbf{1} + \mathbf{1}$. A displayed $(\mathbf{1}+\mathbf{1})$-algebra over **Bool** is a type $\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type}$ equipped with two terms $\Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}]$ and $\Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]$; initiality states that for any such displayed algebra there is a unique displayed algebra homomorphism $(\mathbf{Bool}, \mathbf{true}, \mathbf{false}) \to (A, a_t, a_f)$:

$$\rho_{\Gamma,A,a_t,a_f} : \{a \in \mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \mid a_t = a[\mathbf{id}.\mathbf{true}] \wedge a_f = a[\mathbf{id}.\mathbf{false}]\} \cong \{\star\}$$

We refrain from restating Slogan 2.5.3 in terms of initial algebras, because the general theory of displayed algebras and homomorphisms for a given signature is too significant a detour for these notes; we hope that the reader is convinced that a general pattern exists.

Exercise 2.30. *In Section 2.5.2, our definition of* **Bool** *roughly asserted a natural isomorphism between* $a \in \mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$ *and pairs of substituted terms* $(a[\mathbf{id}.\mathbf{true}], a[\mathbf{id}.\mathbf{false}])$. *Prove that this definition is equivalent to the* ρ_{Γ,A,a_t,a_f} *characterization above.*

2.5.4 *Unicity via extensional equality*

In this section we have defined the inductive types **Void**, **Bool**, and **Nat** by equipping them with constructors and asserting that dependent maps out of them are *judgmentally uniquely determined* by where they send those constructors. That is, a choice of where to send the constructors determines a map via elimination, and any two maps out of an inductive type are judgmentally equal if they agree on the constructors.

This unicity condition is incredibly strong. First of all, it implies the substitution rule for eliminators, because e.g. $\mathbf{if}(a_t, a_f, \mathbf{q})[\gamma.\mathbf{Bool}]$ and $\mathbf{if}(a_t[\gamma], a_f[\gamma], \mathbf{q})$ agree on **true** and **false** (see Exercise 2.26). More alarmingly, in the case of **Void**, it states that *all* terms in contexts containing **Void** are equal to one another (see Exercise 2.25).

It turns out that these unicity principles—the η rules of inductive types—are derivable from the other rules of inductive types in the presence of equality reflection (Section 2.4.4), the other suspiciously strong rule of extensional type theory. For instance:

Theorem 2.5.8. *The following rule (η for* **Void***) can be derived from the other rules for* **Void** *in conjunction with the rules for* **Eq**.

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash a : A}{\Gamma \vdash \mathbf{absurd}(b) = a[\mathbf{id}.b] : A[\mathbf{id}.b]} ✎,\Rightarrow$$

Proof. Suppose $\Gamma \vdash b : \mathbf{Void}$ and $\Gamma.\mathbf{Void} \vdash a : A$. By equality reflection (Section 2.4.4), it suffices to exhibit an element of $\mathbf{Eq}(A[\mathbf{id}.b], \mathbf{absurd}(b), a[\mathbf{id}.b])$, which we obtain easily by **Void** elimination:

$$\Gamma \vdash \mathbf{absurd}(b) : \mathbf{Eq}(A, \mathbf{absurd}(b), a[\mathbf{id}.b]) \qquad \square$$

In Chapter 3 we will see that all of these suspicious rules are problematic from an implementation perspective, leading us to replace extensional type theory with *intensional type theory* (Chapter 4), which differs formally in only two ways: it replaces **Eq**-types with a different equality type that does not admit equality reflection, and it deletes the η rules from **Void**, **Bool**, and **Nat**.

However, in light of the fact that the latter rules are derivable from the former, we—as is conventional—simply omit the η rules for inductive types from the official specification of extensional type theory. (These rules were all marked as provisional ✎.) Note that this does *not* apply to the η rules for $\mathbf{\Pi}$, Σ, **Eq**, or **Unit**, which remain in both type theories.

Semantically, deleting these η rules relaxes the unique existence to simply *existence*. An algebra which admits a (possibly non-unique) algebra homomorphism to any other algebra is known as *weakly initial*, rather than initial. Rather than asking for the collection of algebra homomorphisms to be naturally isomorphic to $\{\star\}$, we simply ask for the map from algebra homomorphisms to $\{\star\}$ to admit a natural *section* (right inverse).

Advanced Remark 2.5.9. Recalling Remark 2.5.4, Theorem 2.5.8 corresponds to the fact that a class of morphisms $\mathcal{L}$ which is weakly orthogonal to $\mathcal{R}$ is actually orthogonal to $\mathcal{R}$ when the latter is closed under relative diagonals ($X \longrightarrow Y \in \mathcal{R}$ implies $X \longrightarrow X \times_Y X \in \mathcal{R}$). $\diamond$

Exercise 2.31. *Prove that the η rule for* **Bool** *can be derived from the other rules for* **Bool** *in conjunction with the rules for* **Eq**, *by mirroring the proof of Theorem 2.5.8.*

2.6 *Universes:* $\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2, \ldots$

We are nearly finished with our definition of extensional type theory, but what's missing is significant: our theory is still not full-spectrum dependent (in the sense of Section 1.1.2)! That is, we have still not introduced the ability to define a family of types whose head type constructor differs at different indices, such as a **Bool**-indexed family of types which sends **true** to **Nat** and **false** to **Unit**. A more subtle but fatal flaw with our current theory is that—despite all the logical connectives at our disposal—we cannot prove that **true** and **false** are different, i.e., we cannot exhibit a term $\mathbf{1} \vdash p : \mathbf{\Pi}(\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}), \mathbf{Void})$.

It turns out that addressing the former will solve the latter *en passant*, so in this section we will discuss two approaches for defining dependent types by case analysis. In Section 2.6.1 we introduce *large elimination*, which equips inductive types with a second elimination principle targeting type-valued algebras (which send each constructor to a *type*), in addition to their usual elimination principle targeting algebras valued in a single dependent type (which send each constructor to a *term* of that type).

Unfortunately we will see that large elimination has some serious limitations, so it will not be an official part of our definition of extensional type theory. Instead, in Section 2.6.2, we introduce *type universes*, connectives which internalize the judgment $\Gamma \vdash A\,\mathsf{type}$ modulo "size issues." By internalizing types as terms of a universe type, universes reduce the problem of computing *types* by case analysis to the problem of computing *terms* by case analysis, which we solved in Section 2.5. That said, universes are a deep and complex topic that will bring us one step closer to our discussion of homotopy type theory in Chapter 5.

2.6.1 *Large elimination*

In Section 2.5 we introduced elimination principles for inductive types (like **Bool**), which allow us to define dependent functions out of an inductive type ($f : \mathbf{\Pi}(\mathbf{Bool}, A)$) by cases on that type's constructors. A direct but uncommon way of achieving full-spectrum dependency is to equip each inductive type with a second elimination principle, *large elimination*, which allows us to define dependent *families of types* by cases.[12]

In the case of **Bool**, large elimination is characterized by the following rules:

$$\frac{\Gamma \vdash A_t \ \mathsf{type} \qquad \Gamma \vdash A_f \ \mathsf{type} \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{If}(A_t, A_f, b) \ \mathsf{type}} \ \text{✎}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A_t \ \mathsf{type} \qquad \Gamma \vdash A_f \ \mathsf{type} \qquad \Gamma \vdash b : \mathbf{Bool}}{\Delta \vdash \mathbf{If}(A_t, A_f, b)[\gamma] = \mathbf{If}(A_t[\gamma], A_f[\gamma], b[\gamma]) \ \mathsf{type}} \ \text{✎}$$

$$\frac{\Gamma \vdash A_t \ \mathsf{type} \qquad \Gamma \vdash A_f \ \mathsf{type}}{\Gamma \vdash \mathbf{If}(A_t, A_f, \mathbf{true}) = A_t \ \mathsf{type} \qquad \Gamma \vdash \mathbf{If}(A_t, A_f, \mathbf{false}) = A_f \ \mathsf{type}} \ \text{✎}$$

If we compare these to the rules of ordinary ("small") elimination,

$$\frac{\Gamma.\mathbf{Bool} \vdash A \ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}] \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}(a_t, a_f, b) : A[\mathbf{id}.b]}$$

$$\frac{\Gamma.\mathbf{Bool} \vdash A \ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}(a_t, a_f, \mathbf{true}) = a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash \mathbf{if}(a_t, a_f, \mathbf{false}) = a_f : A[\mathbf{id}.\mathbf{false}]}$$

we see that the large eliminator **If** is exactly analogous to the small eliminator **if** "specialized to $A :=$ type." Note that this statement is nonsense because the judgment "type" is not a type, but the intuition is useful and will be formalized momentarily. (Indeed, for this reason we cannot formally obtain **If** as a special case of **if**.) Continuing on with the metaphor, the rule for **If** is simpler than the rule for **if** because it has a fixed codomain "type" which is moreover *not* dependent on **Bool**: it makes no sense to ask for "$\Gamma \vdash A_t \ \mathsf{type}[\mathbf{id}.\mathbf{true}]$."

It is even more standard to omit the η rule for large elimination than for small elimination (which is itself typically omitted), but such a rule would state that dependent types indexed by **Bool** are uniquely determined by their values on **true** and **false**:

$$\frac{\Gamma.\mathbf{Bool} \vdash A \ \mathsf{type} \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash A[\mathbf{id}.b] = \mathbf{If}(A[\mathbf{id}.\mathbf{true}], A[\mathbf{id}.\mathbf{false}], b) \ \mathsf{type}} \ \text{✎✎}$$

If we include the η rule, then the rules for **If** would express that instantiating a **Bool**-indexed type at **true** and **false**, i.e. $((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*)$, has a natural inverse:

$$((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*) : \mathsf{Ty}(\Gamma.\mathbf{Bool}) \cong \mathsf{Ty}(\Gamma) \times \mathsf{Ty}(\Gamma)$$

[12] Large elimination maps **Bool** into the collection of all types, which is "large" (in the sense of being "the proper class of all sets") rather than the collection of terms of a single type, which is "small" ("a set").

Again, compare this to our original formulation of small elimination for **Bool**:

$$((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*) : \mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \cong \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}]) \times \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}])$$

When we elide η, large elimination instead states that this map has a *section* (a right inverse), which is to say that a choice of where to send **true** and **false** determines a family of types via **If**, but *not uniquely*. This follows the discussion in Section 2.5.4, except that we *cannot* derive the η rule for large elimination from extensional equality because there is no type "$\mathbf{Eq}(\mathsf{type}, -, -)$" available to carry out the argument in Theorem 2.5.8.

Remark 2.6.1. Large elimination only applies to types defined by mapping-out properties such as inductive types; there is no corresponding principle for mapping-in connectives like $\mathbf{\Pi}(A, B)$ because these do not quantify over any target, whether "small" or "large." ◇

Remark 2.6.2. If we have both small and large elimination for **Bool**, then we can combine them into a derived induction principle for **Bool** that works for any $a_t : A_t$ and $a_f : A_f$, using large elimination to define the type family into which we perform a small elimination.

$$\frac{\Gamma \vdash a_t : A_t \qquad \Gamma \vdash a_f : A_f \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}(a_t, a_f, b) : \mathbf{If}(A_t, A_f, b)} ✎,\Rightarrow$$

◇

With large elimination—or a related feature, type universes—we can prove the disjointness of the booleans. (Although the proof below uses equality reflection, the same theorem holds in intensional type theory for essentially the same reason.) Our claim that we *cannot* prove disjointness without these features is a (relatively simple) independence metatheorem requiring a model construction; see *The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory Without Universes* [Smi88].

Theorem 2.6.3. *Using the rules for* **If***, there is a term*

$$\mathbf{1} \vdash \mathsf{disjoint} : \mathbf{\Pi}(\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}), \mathbf{Void})$$

Proof. We informally describe the derivation of disjoint. By $\mathbf{\Pi}$-introduction we may assume $\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false})$ and prove **Void**. In order to do this, consider the following auxiliary family of types over **Bool**:

$$\mathbf{1}.\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}).\mathbf{Bool} \vdash P \coloneqq \mathbf{If}(\mathbf{Unit}, \mathbf{Void}, \mathbf{q})\ \mathsf{type}$$

Then

$$\begin{aligned} \mathbf{1}.\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}) \vdash \mathbf{Unit} & \\ &= P[\mathbf{id}.\mathbf{true}] && \text{by } \beta \text{ for } \mathbf{If} \\ &= P[\mathbf{id}.\mathbf{false}] && \text{by equality reflection on } \mathbf{q} \\ &= \mathbf{Void}\ \mathsf{type} && \text{by } \beta \text{ for } \mathbf{If} \end{aligned}$$

and therefore $\mathbf{1}.\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}) \vdash \mathbf{tt} : \mathbf{Void}$. In sum, we define $\mathsf{disjoint} \coloneqq \lambda(\mathbf{tt})$. □

As for other inductive types, the large elimination principle of **Void** is:

$$\frac{\Gamma \vdash a : \mathbf{Void}}{\Gamma \vdash \mathbf{Absurd}(a)\ \mathsf{type}}\ ✎$$

Unfortunately, we run into a problem when trying to define large elimination for **Nat**.

$$\frac{\Gamma \vdash n : \mathbf{Nat} \qquad \Gamma \vdash A_z\ \mathsf{type} \qquad \Gamma.\mathbf{Nat}.\text{“type”} \vdash A_s\ \mathsf{type}}{\Gamma \vdash \mathbf{Rec}(A_z, A_s, n)\ \mathsf{type}}\ !?$$

In the ordinary eliminator, the branch for $\mathbf{suc}(-)$ has two variables $m : \mathbf{Nat}, a : A(m)$ binding the predecessor m and (recursively) the result a of the eliminator on m. When “$A :=$ type” the recursive result is a *type*, meaning that the $\mathbf{suc}(-)$ branch ought to bind a *type variable*, a concept which is not a part of our theory. This is a serious problem because recursive constructions of types were a major class of examples in Section 1.1.2.

Exercise 2.32. *There is however a* non-recursive *large elimination principle for* **Nat** *which defines a type by case analysis on whether a number is* **zero**. *This principle follows from the rules in this section along with the other rules of extensional type theory; state and define it.*

2.6.2 Universes

Although large elimination is a useful concept, it sees essentially no use in practice. We have just seen one reason: large eliminators cannot be recursive. The standard approach is instead to include *universe types*, which are “types of types,” or types which internalize the judgment $\Gamma \vdash A\ \mathsf{type}$. Using universes, we can recover large elimination as small elimination into a universe; we are also able to express polymorphic type quantification using dependent functions out of a universe.

A universe is a type with no parameters, so its formation rule is once again a natural family of constants $\mathbf{U}_\Gamma \in \mathsf{Ty}(\Gamma)$, or

$$\frac{}{\Gamma \vdash \mathbf{U}\ \mathsf{type}} \qquad\qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{U} = \mathbf{U}[\gamma]\ \mathsf{type}}$$

As for its terms, the most straightforward definition would be to stipulate a natural isomorphism between terms of **U** and types:

$$\iota : \mathsf{Tm}(\Gamma, \mathbf{U}) \cong \mathsf{Ty}(\Gamma) \tag{?!}$$

Note that just as we did not ask for terms of **Π**-types to literally be terms with an extra free variable, we cannot ask for terms of **U** to literally be types: these are two different sorts!

In inference rules, the forward map of the isomorphism would introduce a new type former $\mathbf{El}(-)$[13] which "decodes" an element of $\mathbf{U}$ into a genuine type. The reverse map conversely "encodes" a genuine type as an element of $\mathbf{U}$. These intuitions lead us to often refer to elements of $\mathbf{U}$ as *codes* for types.

$$\frac{\Gamma \vdash a : \mathbf{U}}{\Gamma \vdash \mathbf{El}(a)\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}}{\Delta \vdash \mathbf{El}(a)[\gamma] = \mathbf{El}(a[\gamma])\ \mathsf{type}}$$

$$\frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash \mathbf{code}(A) : \mathbf{U}}\ ?! \qquad \frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash \mathbf{El}(\mathbf{code}(A)) = A\ \mathsf{type}}\ ?! \qquad \frac{\Gamma \vdash c : \mathbf{U}}{\Gamma \vdash \mathbf{code}(\mathbf{El}(c)) = c : \mathbf{U}}\ ?!$$

Unfortunately we can't have nice things, as the last three rules above—the ones involving **code**—are unsound. In particular they imply that $\mathbf{U}$ contains (a code for) $\mathbf{U}$, making it a "type of all types, including itself" and therefore subject to a variation on Russell's paradox known as *Girard's paradox* [Coq86].

2.6.2.1 Girard's paradox

We present a simplification of Girard's paradox due to Hurkens [Hur95].[14] The details of this paradox are not relevant to any later material in these lecture notes, so the reader may freely skip ahead to Page 58. In this subsection alone, we consider the rules concerning **code** as part of our type theory.

At a high level, the fact that $\mathbf{U}$ contains a code for itself means that we can construct a universe Θ that admits an embedding from its own double power set $\mathcal{P}\ (\mathcal{P}\ \Theta)$; from this we can define a "set of all ordinals" and carry out a version of the Burali-Forti paradox. The details become somewhat involved, in part because the standard paradoxes of set theory rely on comprehension and extensionality principles not available to us in type theory. Indeed, historically it was far from clear that "$\mathbf{U} : \mathbf{U}$" was inconsistent, and in fact Martin-Löf's first version of type theory had this very flaw [ML71].

$$\begin{aligned}
&\mathcal{P} : \mathbf{U} \to \mathbf{U} \\
&\mathcal{P}\ A = \mathbf{code}(\mathbf{El}(A) \to \mathbf{U}) \\[1em]
&\mathcal{P}^2 : \mathbf{U} \to \mathbf{U} \\
&\mathcal{P}^2\ A = \mathcal{P}\ (\mathcal{P}\ A) \\[1em]
&\Theta : \mathbf{U}
\end{aligned}$$

[13] This name is not so mysterious: it means "*elements* of," and is pronounced "ell" or, often, omitted.

[14] An Agda formalization of Hurkens's paradox is available at `https://github.com/agda/agda/blob/master/test/Succeed/Hurkens.agda`; formalizations in other proof assistants are readily available online.

$$\Theta = \mathbf{code}((A : \mathbf{U}) \to (\mathbf{El}(\mathcal{P}^2\ A) \to \mathbf{El}(A)) \to \mathbf{El}(\mathcal{P}^2\ A))$$

Lemma 2.6.4 (Powerful universe)**.** *The universe* Θ *admits maps*

$$\tau : \mathbf{El}(\mathcal{P}^2\ \Theta) \to \Theta$$
$$\sigma : \Theta \to \mathbf{El}(\mathcal{P}^2\ \Theta)$$

such that

$$(C : \mathbf{El}(\mathcal{P}^2\ \Theta)) \to (\sigma\ (\tau\ C) = \lambda(\phi : \mathbf{El}(\mathcal{P}\ \Theta)) \to C(\phi \circ \tau \circ \sigma))$$

Proof. We define:

$$\begin{aligned}
&\tau : \mathbf{El}(\mathcal{P}^2\ \Theta) \to \mathbf{El}(\Theta)\\
&\tau\ (\Phi : \mathbf{El}(\mathcal{P}^2\ \Theta))\ (A : \mathbf{U})\ (f : \mathbf{El}(\mathcal{P}^2\ A) \to \mathbf{El}(A))\ (\chi : \mathbf{El}(\mathcal{P}\ A)) =\\
&\quad \Phi\ (\lambda(\theta : \Theta) \to \chi\ (f\ (\theta\ A\ f)))
\end{aligned}$$

$$\begin{aligned}
&\sigma : \mathbf{El}(\Theta) \to \mathbf{El}(\mathcal{P}^2\ \Theta)\\
&\sigma\ \theta = \theta\ \Theta\ \tau
\end{aligned}$$

We leave the equational condition to Exercise 2.33. □

Exercise 2.33. *Show that the above definitions of* τ *and* σ *satisfy the necessary equation.*

As an immediate consequence of Lemma 2.6.4, we have:

$$\sigma\ (\tau\ (\sigma\ x)) = \lambda(\phi : \mathbf{El}(\mathcal{P}\ \Theta)).\ \sigma\ x\ (\phi \circ \tau \circ \sigma) \tag{2.1}$$

One way to understand the statement of Lemma 2.6.4 is that, regarding $\mathcal{P}$ as a functor whose action on $f : \mathbf{El}(Y) \to \mathbf{El}(X)$ is precomposition $f^* : \mathbf{El}(\mathcal{P}\ X) \to \mathbf{El}(\mathcal{P}\ Y)$, the equational condition is equivalent to $\sigma \circ \tau = (\tau \circ \sigma)^{**}$.

We derive a contradiction from Lemma 2.6.4 by constructing ordinals within Θ:

$$\begin{aligned}
&\text{-- } y < x\ (\text{“}y \in x\text{”})\text{ when each } f \text{ in } \sigma\ x \text{ contains } y\\
&(<) : \mathbf{El}(\Theta) \to \mathbf{El}(\Theta) \to \mathbf{U}\\
&y < x = \mathbf{code}((f : \mathbf{El}(\mathcal{P}\ \Theta)) \to \mathbf{El}(\sigma\ x\ f) \to \mathbf{El}(f\ y))
\end{aligned}$$

$$\begin{aligned}
&\text{-- } f \text{ is inductive if for all } x\text{, if } f \text{ is in } \sigma\ x \text{ then } x \text{ is in } f\\
&\mathsf{ind} : \mathbf{El}(\mathcal{P}\ \Theta) \to \mathbf{U}\\
&\mathsf{ind}\ f = \mathbf{code}((x : \mathbf{El}(\Theta)) \to \mathbf{El}(\sigma\ x\ f) \to \mathbf{El}(f\ x))
\end{aligned}$$

$$\begin{aligned}
&\text{-- } x \text{ is well-founded if it is in every inductive } f\\
&\mathsf{wf} : \mathbf{El}(\Theta) \to \mathbf{U}\\
&\mathsf{wf}\ x = \mathbf{code}((f : \mathbf{El}(\mathcal{P}\ \Theta)) \to \mathbf{El}(\mathsf{ind}\ f) \to \mathbf{El}(f\ x))
\end{aligned}$$

Specifically, we consider $\Omega \coloneqq \tau\ (\lambda f \to \mathsf{ind}\ f)$, the collection of all inductive collections. Using Lemma 2.6.4 we argue that Ω is both well-founded and not well-founded.

Lemma 2.6.5. Ω *is well-founded.*

Proof. Suppose $f : \mathbf{El}(\mathcal{P}\ \Theta)$ is inductive; we must show $\mathbf{El}(f\ \Omega)$. By the definition of ind, for this it suffices to show $\mathbf{El}(\sigma\ \Omega\ f)$. Unfolding the definition of Ω and rewriting by the equation in Lemma 2.6.4 with $C \coloneqq \mathsf{ind}$, it suffices to show that $f \circ \tau \circ \sigma$ is inductive.

Thus suppose we are given $x : \mathbf{El}(\Theta)$ such that $\mathbf{El}(\sigma\ x\ (f \circ \tau \circ \sigma))$; we must show $\mathbf{El}(f\ (\tau\ (\sigma\ x)))$. By rewriting $\mathbf{El}(\sigma\ x\ (f \circ \tau \circ \sigma))$ along Equation (2.1), we conclude that $\mathbf{El}(\sigma\ (\tau\ (\sigma\ x))\ f)$. However, by our assumption that f is inductive, this implies $\mathbf{El}(f\ (\tau\ (\sigma\ x)))$, which is what we wanted to show. □

To prove that Ω is not well-founded, we start by showing that the collection of "sets not containing themselves" $\phi \coloneqq \lambda y \to \mathbf{code}(\mathbf{El}(\tau\ (\sigma\ y) < y) \to \mathbf{Void})$ is inductive.

Lemma 2.6.6. ϕ *is inductive.*

Proof. Suppose we are given x such that $\mathbf{El}(\sigma\ x\ \phi)$; we must show $\mathbf{El}(\tau\ (\sigma\ x) < x) \to \mathbf{Void}$. Thus suppose $\mathbf{El}(\tau\ (\sigma\ x) < x)$, which is to say that for any f such that $\mathbf{El}(\sigma\ x\ f)$, we have $\mathbf{El}(f\ (\tau\ (\sigma\ x)))$. Using our hypothesis we may set $f \coloneqq \phi$, from which we conclude $\mathbf{El}(\tau\ (\sigma\ (\tau\ (\sigma\ x))) < \tau\ (\sigma\ x)) \to \mathbf{Void}$. We derive the required contradiction by proving that $\mathbf{El}(\tau\ (\sigma\ (\tau\ (\sigma\ x))) < \tau\ (\sigma\ x))$ holds, by $\mathbf{El}(\tau\ (\sigma\ x) < x)$ and Exercise 2.34. □

Exercise 2.34. *Show that* $\mathbf{El}(x < y)$ *implies* $\mathbf{El}(\tau\ (\sigma\ x) < \tau\ (\sigma\ y))$.

Theorem 2.6.7. **Void** *is inhabited.*

Proof. Because Ω is well-founded and ϕ is inductive, we have $\mathbf{El}(\tau\ (\sigma\ \Omega) < \Omega) \to \mathbf{Void}$. To derive a contradiction, it suffices to show $\mathbf{El}(\tau\ (\sigma\ \Omega) < \Omega)$, which is to say that for any f such that $\mathbf{El}(\sigma\ \Omega\ f)$, we have $\mathbf{El}(f\ (\tau\ (\sigma\ \Omega)))$. By the definition of Ω, $\mathbf{El}(\sigma\ (\Omega\ f))$ implies that $f \circ \tau \circ \sigma$ is inductive; combining this with the fact that Ω is well-founded, we obtain $\mathbf{El}(f\ (\tau\ (\sigma\ \Omega)))$ as required. □

2.6.2.2 Populating the universe

Returning to our definition of universe types, it is safe to postulate a type $\mathbf{U}$ of type-codes which decode via $\mathbf{El}$ into types. (Indeed, with large elimination it is even possible to define such a type manually, e.g. $\mathbf{U} \coloneqq \mathbf{Bool}$ with $\mathbf{El}(\mathbf{true}) \coloneqq \mathbf{Unit}$ and $\mathbf{El}(\mathbf{false}) \coloneqq \mathbf{Void}$.)

$$\mathbf{U}_\Gamma \in \mathsf{Ty}(\Gamma)$$
$$\mathbf{El} : \mathsf{Tm}(\Gamma, \mathbf{U}) \to \mathsf{Ty}(\Gamma)$$

Our first attempt at populating $\mathsf{Tm}(\Gamma, \mathbf{U})$ was to ask for an inverse to **El**, but that turns out to be inconsistent. Instead, we will simply manually equip **U** with codes decoding to the type formers we have presented so far, but crucially *not* with a code for **U** itself. This approach is somewhat verbose—for each type former we add an introduction rule for **U**, a substitution rule, and an equation stating that **El** decodes it to the corresponding type—but it allows us to avoid Girard's paradox while still populating **U** with codes for (almost) every type in our theory.

For example, to close **U** under dependent function types we add the following rules:

$$\frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Gamma \vdash \mathbf{pi}(a, b) : \mathbf{U}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Delta \vdash \mathbf{pi}(a, b)[\gamma] = \mathbf{pi}(a[\gamma], b[\gamma.\mathbf{El}(a)]) : \mathbf{U}}$$

$$\frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Gamma \vdash \mathbf{El}(\mathbf{pi}(a, b)) = \mathbf{\Pi}(\mathbf{El}(a), \mathbf{El}(b))\ \mathsf{type}}$$

The third rule states that $\mathbf{pi}(a, b)$ is the code in **U** for the type $\mathbf{\Pi}(\mathbf{El}(a), \mathbf{El}(b))$. Note that the context of b in the introduction rule for $\mathbf{pi}(a, b)$ makes reference to $\mathbf{El}(a)$, mirroring the dependency structure of $\mathbf{\Pi}$-types. Although this move is forced, it means that the definitions of **U** and **El** each reference the other—the type of a constructor of **U** mentions **El**, and the type of **El** itself mentions **U**—so **U** and **El** must be defined simultaneously. In fact, this is the paradigmatic example of an *inductive-recursive* definition, an inductive type that is defined simultaneously with a recursive function out of it [Dyb00].

It is no more difficult to close **U** under dependent pairs, extensional equality, the unit type, and inductive types. These rules quickly become tedious, so we write only their introduction rules below, leaving the remaining rules to Appendix A.

$$\frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Gamma \vdash \mathbf{sig}(a, b) : \mathbf{U}} \qquad \frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma \vdash x, y : \mathbf{El}(a)}{\Gamma \vdash \mathbf{eq}(a, x, y) : \mathbf{U}}$$

$$\frac{}{\Gamma \vdash \mathbf{unit} : \mathbf{U}} \qquad \frac{}{\Gamma \vdash \mathbf{nat} : \mathbf{U}} \qquad \frac{}{\Gamma \vdash \mathbf{void} : \mathbf{U}} \qquad \frac{}{\Gamma \vdash \mathbf{bool} : \mathbf{U}}$$

We can now recover the large elimination principles of Section 2.6.1 in terms of small elimination into the type **U**. Moreover, because we can perfectly well extend the context by a variable of type **U**, we can now also construct types by recursion on natural numbers:

$$\frac{\Gamma \vdash n : \mathbf{Nat} \qquad \Gamma \vdash a_z : \mathbf{U} \qquad \Gamma.\mathbf{Nat}.\mathbf{U} \vdash a_s : \mathbf{U}}{\Gamma \vdash \mathbf{Rec}(a_z, a_s, n) \coloneqq \mathbf{El}(\mathbf{rec}(a_z, a_s, n))\ \mathsf{type}} \Rightarrow$$

Remark 2.6.8. Proof assistant users are very familiar with universes, so such readers may be wondering why they have never seen **El** before. Indeed, proof assistants such as Coq and

Agda treat types and elements of $\mathbf{U}$ as indistinguishable. Historically, much of the literature calls such universes—for which $\mathsf{Tm}(\Gamma, \mathbf{U}) \subseteq \mathsf{Ty}(\Gamma)$—*universes à la Russell*, in contrast to our *universes à la Tarski*, but we find such a subset inclusion to be meta-suspicious.

Instead, we prefer to say that Coq and Agda programs do not expose the notion of type to the user at all, instead consistently referring only to elements of $\mathbf{U}$. This obviates the need for the user to ever write or see $\mathbf{El}$, and the necessary calls to $\mathbf{El}$ can be inserted automatically by the proof assistant in a process known as *elaboration*. ⋄

Remark 2.6.9. Another more semantically natural variation of universes relaxes the judgmental equalities governing $\mathbf{El}$ to *isomorphisms* $\mathbf{El}(\mathbf{pi}(a, b)) \cong \mathbf{\Pi}(\mathbf{El}(a), \mathbf{El}(b))$, producing what are known as *weak universes à la Tarski*. However, our *strict* formulation is more standard and more convenient. ⋄

Advanced Remark 2.6.10. Universes in type theory play a similar role to Grothendieck universes and their categorical counterparts in set theory and category theory. We often refer to types encoded by $\mathbf{U}$ as *small* or $\mathbf{U}$-small, and ask for small types to be closed under various operations. As a result, universes in type theory roughly have the same proof-theoretical strength as strongly inaccessible cardinals. Note, however, that the lack of choice and excluded middle in type theory precludes a naïve comparison between it and ZFC or similar theories. ⋄

2.6.3 *Hierarchies of universes*

Our definition of $\mathbf{U}$ is perfectly correct, but the fact that $\mathbf{U}$ lacks a code for itself means that we cannot recursively define types that mention $\mathbf{U}$. In addition, although we can quantify over "small" types with $\mathbf{\Pi}(\mathbf{U}, -)$, we cannot write any type quantifiers whose domain includes $\mathbf{U}$. We cannot fix these shortcomings directly, but we can mitigate them by defining a *second* universe type $\mathbf{U}_1$ closed under all the same type codes as before *as well as a code for* $\mathbf{U}$, but no code for $\mathbf{U}_1$ itself. The same problem occurs one level up, so we add a third universe $\mathbf{U}_2$ containing codes for $\mathbf{U}$ and $\mathbf{U}_1$ but not $\mathbf{U}_2$, and so forth.

In practice, nearly all applications of type theory require only a finite number of universes, but for uniformity and because this number varies between applications, it is typical to ask for a countably infinite (alternatively, finite but arbitrary) tower of universes each of which contains codes for the smaller ones. (For uniformity we write $\mathbf{U}_0 := \mathbf{U}$.) This collection of $\mathbf{U}_i$ is known as a *universe hierarchy*.

To define an infinite number of types and terms, we must now write *rule schemas*, collections of rules that must be repeated for every (external, not internal) natural number $i > 1$. Each of these rules follows the same pattern in $\mathbf{U}$, with one new feature: $\mathbf{U}_i$ contains a code $\mathbf{uni}_{i,j}$ for $\mathbf{U}_j$ whenever j is strictly smaller than i.

$$\frac{}{\Gamma \vdash \mathbf{U}_i \text{ type}} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_i(a) \text{ type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i}{\Delta \vdash \mathbf{El}_i(a)[\gamma] = \mathbf{El}_i(a[\gamma]) \text{ type}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{pi}_i(a,b) : \mathbf{U}_i \qquad \Gamma \vdash \mathbf{sig}_i(a,b) : \mathbf{U}_i} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}_i(a)}{\Gamma \vdash \mathbf{eq}_i(a,x,y) : \mathbf{U}_i}$$

$$\frac{}{\Gamma \vdash \mathbf{unit}_i : \mathbf{U}_i} \qquad \frac{}{\Gamma \vdash \mathbf{void}_i : \mathbf{U}_i} \qquad \frac{}{\Gamma \vdash \mathbf{bool}_i : \mathbf{U}_i} \qquad \frac{}{\Gamma \vdash \mathbf{nat}_i : \mathbf{U}_i} \qquad \frac{j < i}{\Gamma \vdash \mathbf{uni}_{i,j} : \mathbf{U}_i}$$

Again for uniformity we write $\mathbf{pi}_0(a,b) \coloneqq \mathbf{pi}(a,b)$, etc., and we omit the substitution rules for type codes as well as the type equations explaining how each $\mathbf{El}_i$ computes on codes, such as $\mathbf{El}_i(\mathbf{eq}_i(a,x,y)) = \mathbf{Eq}(\mathbf{El}_i(a),x,y)$ and $\mathbf{El}_i(\mathbf{uni}_{i,j}) = \mathbf{U}_j$.

It is easy to see that the rules for $\mathbf{U}_{i+1}$ are a superset of the rules for $\mathbf{U}_i$: the only difference is the addition of the code $\mathbf{uni}_{i+1,i} : \mathbf{U}_{i+1}$ and codes that mention this code, such as $\mathbf{pi}_{i+1}(\mathbf{uni}_{i+1,i}, \mathbf{uni}_{i+1,i}) : \mathbf{U}_{i+1}$. Thus it should be possible to prove that every closed code of type $\mathbf{U}_i$ has a counterpart of type $\mathbf{U}_{i+1}$ that decodes to the same type, that is, "$\mathbf{U}_i \subsetneq \mathbf{U}_{i+1}$." However, this fact is not visible inside the theory. We have no induction principle for the universe, so we cannot define an "inclusion" function $f : \mathbf{U}_i \to \mathbf{U}_{i+1}$ much less prove that it satisfies $\mathbf{El}_{i+1}(f(a)) = \mathbf{El}_i(a)$. And there is simply no way, external or otherwise, to "lift" a variable of type $\mathbf{U}_i$ to the type $\mathbf{U}_{i+1}$.

We thus equip our universe hierarchy with one final operation: a *lifting* operation that includes elements of $\mathbf{U}_i$ into $\mathbf{U}_{i+1}$, which is compatible with $\mathbf{El}$ and sends type codes of $\mathbf{U}_i$ to their counterparts in $\mathbf{U}_{i+1}$. Such a strict lifting operation allows users to generally avoid worrying about universe levels, because small codes can always be hoisted up to their larger counterparts when needed.

$$\frac{\Gamma \vdash c : \mathbf{U}_i}{\Gamma \vdash \mathbf{lift}_i(c) : \mathbf{U}_{i+1}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i}{\Delta \vdash \mathbf{lift}_i(a)[\gamma] = \mathbf{lift}_i(a[\gamma]) : \mathbf{U}_{i+1}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_{i+1}(\mathbf{lift}_i(a)) = \mathbf{El}_i(a) \text{ type}}$$

The last rule above states that a code and its lift both encode the same type. Recalling that the entire point of a universe hierarchy is to get as close as possible to "$\mathbf{U} : \mathbf{U}$" without being inconsistent, it makes sense to treat lifts as a clerical operation that does not affect the type about which we speak. In addition, this equation is actually needed to state that

lift commutes with codes, such as **pi** (other rules omitted):

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{lift}_i(\mathbf{pi}_i(a, b)) = \mathbf{pi}_{i+1}(\mathbf{lift}_i(a), \mathbf{lift}_i(b)) : \mathbf{U}_{i+1}}$$

Remark 2.6.11. We say a universe hierarchy is *(strictly) cumulative* when it is equipped with **lift** operations that commute (strictly) with codes. Historically the term "cumulativity" often refers to material subset inclusions $\mathsf{Tm}(\Gamma, \mathbf{U}_i) \subseteq \mathsf{Tm}(\Gamma, \mathbf{U}_{i+1})$ but once again such conditions are incompatible with our perspective. ⋄

Remark 2.6.12. There is an equivalent presentation of universe hierarchies known as *universes à la Coquand* in which one stratifies the type judgment itself, and the ith universe precisely internalizes the ith type judgment [Coq13; Coq19; Gra+21; FAM23]. That is, we have sorts $\mathsf{Ty}_i(\Gamma)$ for $i \in \mathbb{N} + \{\top\}$ with $\mathsf{Ty}(\Gamma) :\equiv \mathsf{Ty}_\top(\Gamma)$, and natural isomorphisms $\mathsf{Ty}_i(\Gamma) \cong \mathsf{Tm}(\Gamma, \mathbf{U}_i)$ for $i \in \mathbb{N}$ mediated by $\mathbf{El}_i/\mathbf{code}_i$. This presentation essentially creates a new judgmental structure designed to be internalized by $\mathbf{U}$, and has the concrete benefit of unifying type formation and universe introduction into a single set of rules. ⋄

Exercise 2.35. *Check that the equational rule* $\mathbf{lift}_i(\mathbf{pi}_i(a, b)) = \mathbf{pi}_{i+1}(\mathbf{lift}_i(a), \mathbf{lift}_i(b))$ *above is meta-well-typed. (Hint: you need to use* $\mathbf{El}_{i+1}(\mathbf{lift}_i(a)) = \mathbf{El}_i(a)$*.)*

Exercise 2.36. *We only included lifts from* $\mathbf{U}_i$ *to* $\mathbf{U}_{i+1}$*, rather than from* $\mathbf{U}_i$ *to* $\mathbf{U}_j$ *for every* $i < j$*. Show that the latter notion of lift is derivable for any concrete* $i < j$ *and that it satisfies the expected equations.*

Further reading

The literature on type theory is unfortunately neither notationally nor conceptually coherent, particularly regarding syntax and how it is defined. We summarize a number of important references that most closely match the perspective outlined in these lecture notes; note however that many references will agree in some ways and differ in others.

Historical references Nearly all of the ideas in this chapter can be traced back in some form to the philosopher Per Martin-Löf, whose collected works are available in the GitHub repository `michaelt/martin-lof`. Over the decades, Martin-Löf has considered many different variations on type theory; the closest to our presentation are his notes on substitution calculus [ML92] and the "Bibliopolis book" presenting what is now called extensional type theory [ML84]. For a detailed philosophical exploration of the *judgmental methodology* that types internalize judgmental structure, see his "Siena lectures" [ML96]. Finally, the book *Programming in Martin-Löf's Type Theory* [NPS90] remains one of the best pedagogical introductions to type theory as formulated in Martin-Löf's logical framework.

Syntax of dependent type theory The presentation of type theory most closely aligned to ours can be found in the second author's Ph.D. thesis [Gra23, Chapter 2]. Another valuable reference is Hofmann's *Syntax and Semantics of Dependent Types* [Hof97, Sections 1 and 2], which as the title suggests, presents the syntax of type theory and connects it to semantical interpretations. Hofmann is very careful in his definition of syntax, but the technical details of capture-avoiding substitution and presyntax have largely been supplanted by subsequent work on logical frameworks, so we suggest that readers gloss over these technical details.

Categorical semantics The book *Categories for Types* [Cro94] is a gentle introduction to the categorical semantics of the simply-typed lambda calculus and related theories; Castellan, Clairambault, and Dybjer [CCD20] discuss how to scale from such models to categories with families [Dyb96], the categorical counterpart of the substitution calculus. Readers can consult Hofmann [Hof97] for concrete examples of categories with families. Finally, we recommend Awodey's paper on *natural models* [Awo18] for a more categorically-natural formulation of categories with families, as well as an excellent description of the *local universes* strictification procedure for producing models of dependent type theory from categories with enough structure [LW15].

Logical frameworks In these notes we have attempted to largely sidestep the question of what constitutes a valid collection of inference rules. The mathematics of syntax can and has occupied entire books, but in short, the natural families of constants and isomorphisms considered in this chapter can be formulated precisely in systems known as *logical frameworks*. A good introduction to logical frameworks is the seminal work of Harper, Honsell, and Plotkin [HHP93] on the Edinburgh Logical Framework, in which object-level judgments can be encoded as meta-level types.

For logical frameworks better suited to defining dependent type theory in particular, we refer readers to the *generalized algebraic theories* of [Car86] (or the tutorial on this subject by Sterling [Ste19]), or to *quotient inductive-inductive types* [AK16; KKA19; Kov22]. For logical frameworks specifically designed to accomodate the binding and substitution of dependent type theory, we refer the reader to the Ph.D. theses of Haselwarter [Has21] and Uemura [Uem21].

3

Metatheory and implementation

In Chapter 2 we carefully defined Martin-Löf type theory as a formal mathematical object: a kind of "algebra" of indexed sets (of types and terms) equipped with various operations. We believe this perspective is essential to understanding both the *what* and the *why* of type theory, providing both a precise definition that can be unfolded into inference rules, as well as an explanation of what these rules intend to axiomatize.

This perspective is not, however, how most users of type theory interact with it. Most users of type theory interact with *proof assistants*, software systems for interactively developing and verifying large-scale proofs in type theory. Even when type theorists work on paper rather than on a computer, many of the conveniences of proof assistants bleed into their informal notation. Indeed, in Chapter 1 we used definitions, implicit arguments, data type declarations, and pattern matching without a second thought.

Although these lecture notes focus on theoretical rather than practical considerations, it is impossible to discuss the design space of type theory without discussing the pragmatics of proof assistants, as these have exerted a profound influence on the theory. Our goal in this chapter is to explain how to square our mathematical notion of type theory with (idealized) implementations[1] of type theory, and to discover and unpack the substantial constraints that the latter must place on the former.

In this chapter In Section 3.1 we axiomatize the core functionality of proof assistants in terms of algorithmic elaboration judgments, and outline a basic implementation. In Section 3.2 we continue to refine our implementation, taking a closer look at how the equality judgments of type theory impact elaboration, and the metatheoretic properties we need equality to satisfy. In Section 3.3 we discuss other metatheorems of type theory and their relationship to program extraction. In Section 3.4 we show that the extensional type theory presented in Chapter 2 does not satisfy the metatheoretic properties discussed in Section 3.2, leading us to consider alternative theories in Chapters 4 and 5. Finally, in Section 3.5, we consider how to extend our elaborator to account for definitions.

Goals of the chapter By the end of this chapter, you will be able to:

- Explain why and how we define type-checking in terms of elaboration.
- Define the consistency, canonicity, normalization, and invertibility metatheorems, and identify why each is important.

[1] At the end of this chapter, we provide some pointers to literature and implementations specifically geared to readers interested in learning how to actually implement type theory.

- Explain which metatheorems are disrupted by extensional equality, and sketch why.

3.1 *A judgmental reconstruction of proof assistants*

What exactly is the relation between Agda code (or the code in Chapter 1) and the type theory in Chapter 2? Certainly, Coq and Agda—even without extensions—include many convenience features that the reader would not be surprised to see omitted in a theoretical description of type theory: implicit arguments, typeclasses/instance arguments, libraries, reflection, tactics… For the moment we set aside not only these but even more fundamental features such as data type declarations, pattern matching, and the ability to write definitions, in order to consider the simplest possible "Agda": a *type-checker.* That is, our idealized Agda takes as input two expressions e and τ and *accepts* in the case that e is a closed term of closed type τ, and *rejects* if not.

Slogan 3.1.1. *Proof assistants are fancy type-checkers.*

Remark 3.1.2. For the purposes of these notes, "proof assistant" refers only to proof assistants in the style of Coq, Agda, and Lean. In particular, we will not discuss LCF-style systems [GMW79] such as Nuprl [Con+85] and Andromeda [Bau+24], or systems not based on dependent type theory, such as Isabelle [NPW02] or HOL Light [Har09]. ⋄

Convenience features of proof assistants are generally aimed at making it easier for users to write down the inputs e and τ, perhaps by allowing some information to be omitted and reconstructed mechanically, or even by presenting a totally different interface for building e and τ interactively or from high-level descriptions. We start our investigation with the most generous possible assumptions—in which e and τ contain all the information we might possibly need, including type annotations—and will find that type-checking is already a startlingly complex problem.

Remark 3.1.3. The title of this section is an homage to *A judgmental reconstruction of modal logic* [PD01], an influential article that reconsiders intuitionistic modal logic under the mindset that *types internalize judgmental structure.* ⋄

3.1.1 *Type-checking as elaboration*

In Section 2.1 we emphasized that we do *not* assume that the types and terms of type theory are obtained as the "well-formed" subsets of some collections of possibly-ill-formed *pretypes* or *preterms*, nor do we even assume that they are obtained as "$\beta\eta$-equivalence classes" of well-formed-but-unquotiented terms.

Instead, *types* and *terms* are just the elements of the sets $\mathsf{Ty}(\Gamma)$ and $\mathsf{Tm}(\Gamma, A)$, which are defined in terms of each other and the sets Cx and $\mathsf{Sb}(\Delta, \Gamma)$. When we write e.g. $\lambda(b)$, we

Pretypes	$\tau :=$	`(Pi` τ τ`)` | `(Sigma` τ τ`)` | `Unit` | `Uni` | `(El` e`)` | $\cdots$
Preterms	$e :=$	`(var` i`)` | `(lam` τ τ e`)` | `(app` τ τ e e`)` | `(pair` τ τ e e`)` | `(fst` τ τ e`)` | $\cdots$
Indices	$i :=$	`0` | `1` | `2` | $\cdots$

Figure 3.1: Syntax of pretypes and preterms.

are naming a particular element of a particular set $\mathsf{Tm}(\Gamma, \mathbf{\Pi}(A, B))$ obtained by applying the "$\mathbf{\Pi}$-introduction" map to $b \in \mathsf{Tm}(\Gamma.A, B)$; in particular, the values of Γ, A, B should be regarded as implicitly present, as they are in Appendix A where we write $\lambda_{\Gamma,A,B}(b)$.

In Chapter 2 we reaped the benefits of this perspective, but it has come time to pay the piper: what, then, is a type-checker supposed to take as input? We certainly cannot say that a type-checker is given "a type A and a term a" because this assumes that A and a are well-formed. Type-checking *cannot be a membership query*; instead, it is a *partial function* from concrete syntax to the sets of genuine types and terms. For an input expression to "type-check" means that it *names* a type/term, not that it "is" one (which is a meta-type error, as types/terms are mathematical objects, and input expressions are strings).

For simplicity we assume that the inputs to type-checkers are not strings but abstract syntax trees (or well-formed formulas) conforming to the simple grammar in Figure 3.1.[2] We call these semi-structured input expressions *pretypes* τ and *preterms* e, and write them as `teletype` s-expressions. In programming language theory, the process of mapping semi-structured input expressions into structured core language terms is known as *elaboration*.

Slogan 3.1.4. *Type-checkers for dependent type theory are elaborators.*

Remark 3.1.5. What is the relationship between features of the concrete syntax of a proof assistant, and features of the core syntax? According to Slogan 3.1.4, the concrete syntax should be seen as "instructions" for building core syntax. These instructions may be very close to or very far from that core syntax, but in either case, new user-facing features should only induce new core primitives when they cannot be (relatively compositionally) accounted for by the existing core language. ⋄

Algorithmic judgments Elaborators are partial functions that recursively consume pretypes and preterms (abstract syntax trees) and produce types and terms. In a real proof assistant, types and terms are of course not abstract mathematical entities but elements of some data type, but for our purposes we will imagine an idealized elaborator that outputs elements of $\mathsf{Ty}(\Gamma)$ and $\mathsf{Tm}(\Gamma, A)$. We present this elaborator not as functional programs written in pseudocode, but as *algorithmic judgments* defined by inference rules. Unlike the rules in Chapter 2, these rules are intended to define an algorithm, so we will take care to

[2] In other words, we only consider input expressions that successfully parse; expressions that fail to parse (e.g., because their parentheses are mismatched) automatically fail to type-check.

ensure that any given elaboration judgment can be derived by at most one rule. (In other words, we define our elaborator as a deterministic logic program.)

We have already argued that pretype elaboration should take as input a pretype τ and output a type A, but what about contexts? Just as well-formedness of closed types ($\mathbf{1} \vdash \mathbf{\Pi}(A, B)\,\mathsf{type}$) refers to well-formedness of open types ($\mathbf{1}.A \vdash B\,\mathsf{type}$), it is perhaps unsurprising that elaborating closed pretypes requires elaborating open pretypes. However, we note that we do not need or want "precontexts"; we will only descend under binders after successfully elaborating their pretypes. For example, to elaborate $(\mathtt{Pi}\ \tau_0\ \tau_1)$ we will first elaborate τ_0 to the closed type A, and only then in context $\mathbf{1}.A$ elaborate τ_1 to B.

Thus our two main algorithmic elaboration judgments are as follows:

1. $\Gamma \vdash \tau\,\mathsf{type} \rightsquigarrow A$ asserts that elaborating the pretype τ relative to $\vdash \Gamma\,\mathsf{cx}$ succeeds and produces the type $\Gamma \vdash A\,\mathsf{type}$.

2. $\Gamma \vdash e : A \rightsquigarrow a$ asserts that elaborating the preterm e relative to $\vdash \Gamma\,\mathsf{cx}$ and $\Gamma \vdash A\,\mathsf{type}$ succeeds and produces the term $\Gamma \vdash a : A$.

In pseudocode, the first judgment corresponds to a partial function $\mathtt{elabTy}(\Gamma, \tau) = A$ with the invariant that if $\vdash \Gamma\,\mathsf{cx}$ and $\mathtt{elabTy}$ terminates successfully, then $\Gamma \vdash A\,\mathsf{type}$. Likewise, the second judgment is a partial function $\mathtt{elabTm}(\Gamma, A, e) = a$ whose successful outputs are terms $\Gamma \vdash a : A$.

Elaborating pretypes The rules for $\Gamma \vdash \tau\,\mathsf{type} \rightsquigarrow A$ are straightforward translations of the type-well-formedness rules of Chapter 2. (When it is necessary to contrast algorithmic and non-algorithmic rules, the latter are often referred to as *declarative*.)

$$\frac{\Gamma \vdash \tau_0\,\mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\,\mathsf{type} \rightsquigarrow B}{\Gamma \vdash (\mathtt{Pi}\ \tau_0\ \tau_1)\,\mathsf{type} \rightsquigarrow \mathbf{\Pi}(A, B)} \qquad \frac{\Gamma \vdash \tau_0\,\mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\,\mathsf{type} \rightsquigarrow B}{\Gamma \vdash (\mathtt{Sigma}\ \tau_0\ \tau_1)\,\mathsf{type} \rightsquigarrow \Sigma(A, B)}$$

$$\frac{}{\Gamma \vdash \mathtt{Unit}\,\mathsf{type} \rightsquigarrow \mathbf{Unit}} \qquad \frac{}{\Gamma \vdash \mathtt{Uni}\,\mathsf{type} \rightsquigarrow \mathbf{U}} \qquad \frac{\Gamma \vdash e : \mathbf{U} \rightsquigarrow a}{\Gamma \vdash (\mathtt{El}\ e)\,\mathsf{type} \rightsquigarrow \mathbf{El}(a)}$$

3.1.2 *Elaborating preterms: the problem of type equality*

Elaborating preterms is significantly more fraught. But first, let us remind ourselves of the process of type-checking $(\mathtt{lam}\ \tau_0\ \tau_1\ e) : \tau$. First, we attempt to elaborate the pretype $\mathbf{1} \vdash \tau\,\mathsf{type} \rightsquigarrow C$; if this succeeds, we then attempt to elaborate the preterm $\mathbf{1} \vdash (\mathtt{lam}\ \tau_0\ \tau_1\ e) : C \rightsquigarrow c$. If this also succeeds, then the type-checker reports success, having transformed the input presyntax to a well-formed term $\mathbf{1} \vdash c : C$.

Since `lam` is our presyntax for λ, elaborating `lam` via $\mathbf{1} \vdash (\texttt{lam}\ \tau_0\ \tau_1\ e) : C \rightsquigarrow c$ should produce a term $c :\equiv \lambda_{1,A,B}(b)$ for some A, B, b determined by τ_0, τ_1, e respectively. We determine these by a series of recursive calls to the elaborator: first $\Gamma \vdash \tau_0\ \mathsf{type} \rightsquigarrow A$, then $\Gamma.A \vdash \tau_1\ \mathsf{type} \rightsquigarrow B$, and finally $\Gamma.A \vdash e : B \rightsquigarrow b$. Note that these steps must be performed sequentially and in this order, because each step uses the outputs of the previous steps as inputs: we elaborate τ_1 in a context extended by A, the result of elaborating τ_0, and we elaborate e at type B, the result of elaborating τ_1.

At the end we obtain $\Gamma.A \vdash b : B$, and thence by $\mathbf{\Pi}$-introduction a term $\mathbf{1} \vdash \lambda_{1,A,B}(b) : \mathbf{\Pi}_1(A, B)$ that should be the elaborated form of e. But the elaborated form of e is supposed to have type C—the result of elaborating τ! Thus before returning $\lambda_{1,A,B}(b)$ we need to check that $\mathbf{1} \vdash C = \mathbf{\Pi}(A, B)\ \mathsf{type}$. This is where "type-checking" actually happens: we have seen that τ determines a real type and that e determines a real term, but until this point we have not actually checked whether "e has type τ."

In pseudocode, we can define elaboration of $(\texttt{lam}\ \tau_0\ \tau_1\ e)$ as follows:

$$
\begin{array}{l}
\texttt{elabTm}(\Gamma, C, (\texttt{lam}\ \tau_0\ \tau_1\ e)) = \\
\quad \texttt{let}\ A = \texttt{elabTy}(\Gamma, \tau_0)\ \texttt{in} \\
\quad \texttt{let}\ B = \texttt{elabTy}(\Gamma.A, \tau_1)\ \texttt{in} \\
\quad \texttt{let}\ b = \texttt{elabTm}(\Gamma.A, B, e)\ \texttt{in} \\
\quad \texttt{if}\ (\Gamma \vdash C = \mathbf{\Pi}_\Gamma(A, B)\ \mathsf{type})\ \texttt{then}\ \texttt{return}\ \lambda_{\Gamma,A,B}(b)\ \texttt{else}\ \texttt{error}
\end{array}
$$

or equivalently, in algorithmic judgment notation:

$$
\frac{\Gamma \vdash \tau_0\ \mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\ \mathsf{type} \rightsquigarrow B \qquad \Gamma.A \vdash e : B \rightsquigarrow b \qquad \Gamma \vdash C = \mathbf{\Pi}(A, B)\ \mathsf{type}}{\Gamma \vdash (\texttt{lam}\ \tau_0\ \tau_1\ e) : C \rightsquigarrow \lambda_{\Gamma,A,B}(b)}
$$

This will be the only rule that concludes $\Gamma \vdash e : C \rightsquigarrow c$ for $e :\equiv (\texttt{lam}\ \tau_0\ \tau_1\ e)$, ensuring that this rule "is the `lam` clause of `elabTm`," so to speak. Elaboration of other introduction forms will follow a similar pattern.

Exercise 3.1. *Write the algorithmic rule for elaborating the preterm* $(\texttt{pair}\ \tau_0\ \tau_1\ e_0\ e_1)$.

Let us pause to make several remarks. First, note that our algorithm needs to check judgmental equality of types $\Gamma \vdash C = \mathbf{\Pi}_\Gamma(A, B)\ \mathsf{type}$. This step is, at least implicitly, part of all type-checking algorithms for all programming languages: if we define a function of type $A \to B$ that returns e, we have to check whether the type of e matches the declared return type B. Sometimes this is as simple as checking the syntactic equality of two type expressions, but often this is non-trivial, perhaps a subtyping check.

In our present setting, checking type equality is *extremely* non-trivial. Suppose that $C :\equiv \mathbf{El}(c)$ and so we are checking $\Gamma \vdash \mathbf{El}(c) = \mathbf{\Pi}(A, B)\ \mathsf{type}$ for $\Gamma \vdash c : \mathbf{U}$. This type equality depends on the entire equational theory of *terms*: we may need to "rewrite along" arbitrarily many term equations before concluding $\Gamma \vdash c = \mathbf{pi}(c_0, c_1) : \mathbf{U}$; this only reduces the problem to $\Gamma \vdash \mathbf{\Pi}(\mathbf{El}(c_0), \mathbf{El}(c_1)) = \mathbf{\Pi}(A, B)\ \mathsf{type}$ for which it suffices to check $\Gamma \vdash \mathbf{El}(c_0) = A\ \mathsf{type}$ and $\Gamma.A \vdash \mathbf{El}(c_1) = B\ \mathsf{type}$, each of which may once again require arbitrary amounts of computation. We will revisit this point in Section 3.2.1.

Secondly, note that we have assumed for now that the preterm $(\mathtt{lam}\ \tau_0\ \tau_1\ e)$ contains pretype annotations τ_0, τ_1 telling us the domain and codomain of the $\mathbf{\Pi}$-type. In practice, a type-checker is essentially unusable unless it can *reconstruct* (most of) these annotations; we describe this reconstruction process in Section 3.2.2.

Remark 3.1.6. Naïvely, one might think that *including* these annotations is the source of our problem, because it forces us to compare the type C computed from τ to the type $\mathbf{\Pi}(A, B)$ computed from the annotations τ_0, τ_1. This is not the case. If we omit τ_0, τ_1, then to elaborate e we must *recover* A and B from C, which upgrades "does $\Gamma \vdash C = \mathbf{\Pi}(A, B)\ \mathsf{type}$?" to the strictly harder question "do there *exist* A, B such that $\Gamma \vdash C = \mathbf{\Pi}(A, B)\ \mathsf{type}$?" In addition, we will need to wonder whether this existence is unique: otherwise, it could be that $\Gamma.A \vdash e : B \rightsquigarrow b$ for some choices of A, B but not others. ⋄

Elaborating elimination forms is not much harder than elaborating introduction forms. To elaborate $(\mathtt{app}\ \tau_0\ \tau_1\ e_0\ e_1)$, we elaborate the pretype annotations $\Gamma \vdash \tau_0\ \mathsf{type} \rightsquigarrow A$ and $\Gamma.A \vdash \tau_1\ \mathsf{type} \rightsquigarrow B$ in sequence, then the function $\Gamma \vdash e_0 : \mathbf{\Pi}(A, B) \rightsquigarrow f$ and its argument $\Gamma \vdash e_1 : A \rightsquigarrow a$ in either order, before finally checking that the type of the computed term $\mathbf{app}_{\Gamma,A,B}(f, a)$, namely $B[\mathbf{id}.a]$, agrees with the expected type C.

$$\frac{\begin{array}{c}\Gamma \vdash \tau_0\ \mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\ \mathsf{type} \rightsquigarrow B \\ \Gamma \vdash e_0 : \mathbf{\Pi}(A, B) \rightsquigarrow f \qquad \Gamma \vdash e_1 : A \rightsquigarrow a \qquad \Gamma \vdash C = B[\mathbf{id}.a]\ \mathsf{type}\end{array}}{\Gamma \vdash (\mathtt{app}\ \tau_0\ \tau_1\ e_0\ e_1) : C \rightsquigarrow \mathbf{app}_{\Gamma,A,B}(f, a)}$$

Elaboration of other elimination forms follows a similar pattern. The only remaining case is term variables $(\mathtt{var}\ i)$, which we have chosen to represent as de Bruijn indices. To elaborate $(\mathtt{var}\ i)$ we check that the context has length at least $i + 1$; if so, then it remains only to check that the type of the variable $\mathbf{q}[\mathbf{p}^i]$ agrees with the expected type.

$$\frac{\Gamma = \Gamma'.A_i.A_{i-1}.\cdots.A_0 \qquad \Gamma \vdash C = A_i[\mathbf{p}^i]\ \mathsf{type}}{\Gamma \vdash (\mathtt{var}\ i) : C \rightsquigarrow \mathbf{q}[\mathbf{p}^i]}$$

Remark 3.1.7. It is straightforward to extend our concrete syntax to support named variables: in our elaboration judgments, we replace Γ with an *environment* Θ that is a list of pairs of genuine types with the "surface name" of the corresponding term variable. Every environment determines a context by forgetting the names; in the variable elaboration rule, we simply look up the de Bruijn index corresponding to the given name. ⋄

Exercise 3.2. *Write the algorithmic rules for elaborating* $(\mathtt{fst}\ \tau_0\ \tau_1\ e)$ *and* $(\mathtt{snd}\ \tau_0\ \tau_1\ e)$.

3.2 *Metatheory for type-checking*

In Section 3.1 we saw that we can reduce type-checking to the problem of deciding the equality of types (at least, assuming that our input preterms have all type annotations). Deciding the equality of types in turn requires deciding the equality of terms, particularly in the presence of universes (Section 2.6.2). It is far from obvious that these relations are decidable—in fact, as we will see in Section 3.4, they are actually *undecidable* for the theory presented in Chapter 2—and proving their decidability relies on a difficult metatheorem known as *normalization*. In this section, we continue our exploration of elaboration with an emphasis on normalization and other metatheorems necessary for type-checking.

Remark 3.2.1. Recall from Section 2.1 that a *metatheorem* is just an ordinary theorem in the ambient metatheory, particularly one concerning the object type theory. ⋄

Before we can discuss computability-theoretic properties of the judgments of type theory, however, we must fix an encoding. We have taken pains to treat the rules of type theory as defining abstract sets $\mathsf{Ty}(\Gamma)$ and $\mathsf{Tm}(\Gamma, A)$ equipped with functions (type and term formers) satisfying various equations (β and η laws), which is the right perspective for understanding the mathematical structure of type theory. But to discuss the *computational* properties of type theory it is essential to exhibit an effective encoding of types and terms that is suitable for manipulation by a Turing machine or other model of computation: Turing machines cannot take mathematical entities as inputs, and whether equality of types is decidable can depend on how we choose to encode them!

This is analogous to the issue that arises in elementary computability theory when formalizing the halting problem: we must agree on how to encode Turing machines as inputs to other Turing machines, and we must ensure that this encoding is suitably effective. It is possible to pick an encoding of computable functions that trivializes the halting problem, at the expense of this encoding itself necessarily being uncomputable.

Returning to type theory, derivation trees of inference rules (e.g., as in Appendix A) turn out to be a perfectly suitable encoding. That is, when discussing computability-theoretic properties of types, terms, and equality judgments, we shall assume that each of these is encoded by equivalence classes of closed derivation trees; for example, we encode $\mathsf{Ty}(\Gamma)$ by the set of derivation trees with root $\Gamma \vdash A\ \mathsf{type}$ for some A. (Just as there are many Turing machines realizing any given function $\mathbb{N} \to \mathbb{N}$, there will be many derivation trees encoding any given type $A \in \mathsf{Ty}(\Gamma)$.) When we say "equality of types is decidable," what we shall mean is that "it is decidable whether two derivations encode the same type." But having fixed a convention, we will avoid belaboring the point any further.

3.2.1 *Normalization and the decidability of equality*

To complete the pretype and preterm elaboration algorithms presented in Section 3.1, it remains only to show that type and term equality are decidable, which is equivalent to the following normalization condition.

Remark 3.2.2. Type and term equality are automatically *semidecidable* because derivation trees are recursively enumerable. That is, to check whether two types $A, B \in \mathsf{Ty}(\Gamma)$ are equal, we can enumerate every derivation tree of type theory, terminating if we encounter a derivation of $\Gamma \vdash A = B$ type. Obviously, this is not a realistic implementation strategy. ⋄

Definition 3.2.3. A *normalization structure* for a type theory is a pair of computable, injective functions $\mathtt{nfTy} : \mathsf{Ty}(\Gamma) \to \mathbb{N}$ and $\mathtt{nfTm} : \mathsf{Tm}(\Gamma, A) \to \mathbb{N}$.

Definition 3.2.4. A type theory enjoys *normalization* if it admits a normalization structure.

The reader may find these definitions surprising: where did $\mathbb{N}$ come from, and where is the rest of the definition? We have chosen $\mathbb{N}$ because it is a countable set with decidable equality, but any other such set would suffice. In practice, one instead defines two sets of abstract syntax trees $\mathtt{TyNf}$, $\mathtt{TmNf}$ with discrete equality, and constructs a pair of computable, injective functions $\mathtt{nfTy} : \mathsf{Ty}(\Gamma) \to \mathtt{TyNf}$ and $\mathtt{nfTm} : \mathsf{Tm}(\Gamma, A) \to \mathtt{TmNf}$. It is trivial to exhibit computable, injective Gödel encodings of $\mathtt{TyNf}$ and $\mathtt{TmNf}$, which when composed with $\mathtt{nfTy}, \mathtt{nfTm}$ exhibit a normalization structure in the sense of Definition 3.2.3.

As for Definition 3.2.3 being sufficient, the force of normalization is that it gives us a decision procedure for type/term equality as follows: given $A, B \in \mathsf{Ty}(\Gamma)$, A and B are equal if and only if $\mathtt{nfTy}(A) = \mathtt{nfTy}(B)$ in $\mathbb{N}$. Asking for these maps to be computable ensures that this procedure is computable; injectivity ensures that it is *complete* in the sense that $\mathtt{nfTy}(A) = \mathtt{nfTy}(B)$ implies $A = B$. The *soundness* of this procedure—that $A = B$ implies $\mathtt{nfTy}(A) = \mathtt{nfTy}(B)$—is implicit in the statement that $\mathtt{nfTy}$ is a function out of $\mathsf{Ty}(\Gamma)$, the set of types considered modulo judgmental equality.

Warning 3.2.5. In Section 3.4 we shall see that extensional type theory *does not* admit a normalization structure, but we will proceed under the assumption that the theory we are elaborating satisfies normalization. In Chapter 4 we will see how to modify our type theory to substantiate this assumption.

Assuming normalization, we can define algorithmic type and term equality judgments

1. $\Gamma \vdash A \Leftrightarrow B$ type asserts that the types $\Gamma \vdash A$ type and $\Gamma \vdash B$ type are judgmentally equal according to some decision procedure.

2. $\Gamma \vdash a \Leftrightarrow b : A$ asserts that the terms $\Gamma \vdash a : A$ and $\Gamma \vdash a : A$ are judgmentally equal according to some decision procedure.

as follows:

$$\frac{\mathtt{nfTy}(A) = \mathtt{nfTy}(B)}{\Gamma \vdash A \Leftrightarrow B\,\mathsf{type}} \qquad \frac{\mathtt{nfTm}(a) = \mathtt{nfTm}(b)}{\Gamma \vdash a \Leftrightarrow b : A}$$

We notate algorithmic equality differently from the declarative equality judgments $\Gamma \vdash A = B\,\mathsf{type}$ and $\Gamma \vdash a = b : A$ to stress that their definitions are completely different, even though (by our argument above) two types/terms are algorithmically equal if and only if they are declaratively equal. We thus complete the elaborator from Section 3.1 by replacing the "calls" to $\Gamma \vdash C = \mathbf{\Pi}(A, B)\,\mathsf{type}$ with calls to $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A, B)\,\mathsf{type}$.

Remark 3.2.6. It may seem surprising that normalization is so difficult; why can't algorithmic equality just *orient* each declarative equality rule (e.g., $\mathbf{fst}(\mathbf{pair}(a, b)) \rightsquigarrow a$) and check whether the resulting rewriting system is confluent and terminating? Unfortunately, while this strategy suffices for some dependent type theories such as the calculus of constructions [CH88], it is very difficult to account for judgmental η rules. (What direction should $p \leftrightsquigarrow \mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p))$ go? What about the η rule of $\mathbf{Unit}$, $a \leftrightsquigarrow \mathbf{tt}$?) These rules require a type-sensitive decision procedure known as *normalization by evaluation*, whose soundness and completeness for declarative equality is nontrivial [ACD07; Abe13]. ⋄

Exercise 3.3. *We argued that the existence of a normalization structure implies that judgmental equality is decidable. In fact, this is a biimplication. Assume that definitional equality is decidable, and construct from this a normalization structure. (Hint: some classical reasoning is required, such as Markov's principle or the law of excluded middle.)*

Exercise 3.4. *We have sketched how to use normalization to obtain a type-checking algorithm. This, too, is a biimplication. Using Exercise 3.3, show that the ability to decide type-checking implies that normalization holds.*

3.2.2 *Injectivity and bidirectional type-checking*

We have seen how to define a rudimentary elaborator for type theory assuming that normalization holds, but the preterms that we can elaborate (Figure 3.1) are quite verbose, making our proof assistant more of a proof adversary. For instance, function application ($\mathsf{app}\ \tau_0\ \tau_1\ e_0\ e_1$) requires annotations for both the domain and codomain of the $\mathbf{\Pi}$-type.

These annotations are highly redundant, but it is far from clear how many of them can be mechanically reconstructed by our elaborator, nor if there is a consistent strategy for doing so. Users of typed functional programming languages like OCaml or Haskell might imagine that virtually all types can be inferred automatically; unfortunately, this is impossible in dependent type theory, for which type inference is undecidable [Dow93].

It turns out there is a fairly straightforward, local, and usable approach to type reconstruction known as *bidirectional type-checking* [Coq96; PT00; McB18; McB19]. The core

$$\begin{array}{lll} \textit{Pretypes} & \tau \coloneqq & (\texttt{Pi}\ \tau\ \tau) \mid (\texttt{Sigma}\ \tau\ \tau) \mid \texttt{Unit} \mid \texttt{Uni} \mid (\texttt{El}\ e) \mid \cdots \\ \textit{Preterms} & e \coloneqq & (\texttt{var}\ i) \mid (\texttt{chk}\ e\ \tau) \mid (\texttt{lam}\ e) \mid (\texttt{app}\ e\ e) \mid (\texttt{pair}\ e\ e) \mid (\texttt{fst}\ e) \mid \cdots \end{array}$$

Figure 3.2: Syntax of pretypes and preterms for a bidirectional elaborator.

insight of bidirectional type-checking is that for some preterms it is easy to reconstruct or *synthesize* its type (e.g., if we know a function's type then we know the type of its applications), but for other preterms we must be given a type at which to *check* it (e.g., to type-check a function we need to be told the type of its input variable).

By explicitly splitting elaboration into two mutually-defined algorithms—type-checking and type synthesis—we can dramatically reduce type annotations. In fact, in Figure 3.2 we can see that our new preterm syntax has no type annotations whatsoever except for a single annotation form $(\texttt{chk}\ e\ \tau)$ that we will use sparingly. The ebb and flow of information between terms and types—between checking and synthesis—leads to the eponymous bidirectional flow of information that has proven easily adaptable to new type theories. But when should we check, and when should we synthesize?

Slogan 3.2.7. *Types are checked in introduction rules, and synthesized in elimination rules.*

We replace our two algorithmic elaboration judgments $\Gamma \vdash \tau\ \mathsf{type} \rightsquigarrow A$ and $\Gamma \vdash e : A \rightsquigarrow a$ with three algorithmic judgments as follows:

1. $\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A$ ("check τ") asserts that elaborating the pretype τ relative to $\vdash \Gamma\ \mathsf{cx}$ succeeds and produces the type $\Gamma \vdash A\ \mathsf{type}$.

2. $\Gamma \vdash e \Leftarrow A \rightsquigarrow a$ ("check e against A") asserts that elaborating the (unannotated) preterm e relative to $\vdash \Gamma\ \mathsf{cx}$ and a *given* type $\Gamma \vdash A\ \mathsf{type}$ succeeds with $\Gamma \vdash a : A$.

3. $\Gamma \vdash e \Rightarrow A \rightsquigarrow a$ ("synthesize A from e") asserts that elaborating the (unannotated) preterm e relative to $\vdash \Gamma\ \mathsf{cx}$ succeeds and *produces* both $\Gamma \vdash A\ \mathsf{type}$ and $\Gamma \vdash a : A$.

The first two judgments, $\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A$ and $\Gamma \vdash e \Leftarrow A \rightsquigarrow a$, are similar to our previous judgments; when elaborating a preterm we are given a context and a type at which to check that preterm. In the third judgment, $\Gamma \vdash e \Rightarrow A \rightsquigarrow a$, we are also given a preterm and a context, but we output *both a term and its type*. The arrows are meant to indicate the direction of information flow: when checking $e \Leftarrow A$ we are given A and using it to elaborate e, but when synthesizing $e \Rightarrow A$ we are producing A from e.

The rules for $\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A$ are the same as those for $\Gamma \vdash \tau\ \mathsf{type} \rightsquigarrow A$, except that they reference the new checking judgment $\Gamma \vdash e \Leftarrow A \rightsquigarrow a$ instead of $\Gamma \vdash e : A \rightsquigarrow a$. But for each old $\Gamma \vdash e : A \rightsquigarrow a$ rule, we must decide whether this preterm should be checked or synthesized, and if the latter, how to reconstruct the type.

The easiest case is the variable $(\texttt{var}\ i)$. Elaboration always takes place with respect to a context which records the types of each variable, so it is easy to synthesize the variable's type. Notably, unlike in our previous variable rule, we do not need to check type equality!

$$\frac{\Gamma = \Gamma'.A_i.A_{i-1}.\cdots.A_0}{\Gamma \vdash (\texttt{var}\ i) \Rightarrow A_i[\mathbf{p}^i] \rightsquigarrow \mathbf{q}[\mathbf{p}^i]}$$

Next, let us consider the rules for $\mathbf{\Pi}$-types. According to Slogan 3.2.7, the introduction form $(\texttt{lam}\ e)$ should be checked. As in Section 3.1, to check $\Gamma \vdash (\texttt{lam}\ e) \Leftarrow C \rightsquigarrow \lambda(b)$ we must recursively check the body of the lambda, $\Gamma.A \vdash e \Leftarrow B \rightsquigarrow b$. But where do A and B come from? (Last time, we elaborated them from `lam`'s annotations.) We might imagine that we can recover A and B from the given type C,

$$\frac{\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A, B)\ \mathsf{type} \qquad \Gamma.A \vdash e \Leftarrow B \rightsquigarrow b}{\Gamma \vdash (\texttt{lam}\ e) \Leftarrow C \rightsquigarrow \lambda(b)}\ !?$$

but this rule does not make sense as written; $\Gamma \vdash C \Leftrightarrow D\ \mathsf{type}$ is an algorithm which takes two types and returns "yes" or "no", and we cannot use it to invent the types A and B.

Worse yet, as foreshadowed in Remark 3.1.6, even if we can find A and B such that $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A, B)\ \mathsf{type}$, there is no reason to expect this choice to be unique. That is, it could be that $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A, B)\ \mathsf{type}$ and $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A', B')\ \mathsf{type}$ both hold, but $A \neq A'$ (or alternatively, $A = A'$ and $B \neq B'$). If so, it is possible that e elaborates with respect to one of these choices but not the other, i.e., $\Gamma.A \vdash e \Leftarrow B \rightsquigarrow b$ succeeds but $\Gamma.A' \vdash e \Leftarrow B' \rightsquigarrow\ ?$ fails; even if both succeed, they will necessarily elaborate two different terms! We must foreclose these possibilities in order for elaboration to be well-defined.

Definition 3.2.8. A type theory has *injective $\mathbf{\Pi}$-types* if $\Gamma \vdash \mathbf{\Pi}(A, B) = \mathbf{\Pi}(A', B')\ \mathsf{type}$ implies $\Gamma \vdash A = A'\ \mathsf{type}$ and $\Gamma.A \vdash B = B'\ \mathsf{type}$.

Definition 3.2.9. A type theory has *invertible $\mathbf{\Pi}$-types* if it has injective $\mathbf{\Pi}$-types and admits a computable function which, given $\Gamma \vdash C\ \mathsf{type}$, either produces the unique $\Gamma \vdash A\ \mathsf{type}$ and $\Gamma.A \vdash B\ \mathsf{type}$ for which $\Gamma \vdash C = \mathbf{\Pi}(A, B)\ \mathsf{type}$, or determines that no such A, B exist.

Remark 3.2.10. That is, a type theory has injective $\mathbf{\Pi}$-types if the type former $\mathbf{\Pi}_\Gamma : (\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)) \to \mathsf{Ty}(\Gamma)$ is injective. A type theory has invertible $\mathbf{\Pi}$-types if the image of $\mathbf{\Pi}_\Gamma$ is decidable and $\mathbf{\Pi}_\Gamma$ admits a (computable) partial inverse $\mathbf{\Pi}_\Gamma^{-1} : \mathrm{Im}(\mathbf{\Pi}_\Gamma) \to (\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A))$. ⋄

Particularly in light of Remark 3.2.10, one can easily extend the terminology of injectivity and invertibility to non-$\mathbf{\Pi}$ type formers.

Definition 3.2.11. If all the type constructors of a type theory are injective (resp., invertible), we say that the type theory *has injective (resp., invertible) type constructors.*

Having injective or invertible type constructors does not follow from normalization. (A type theory in which all types and terms are equal to each other is normalizing but not injective!) In practice, however, having invertible type constructors is almost always an immediate consequence of the *proof* of normalization. As we mentioned in Section 3.2.1, normalization proofs generally construct abstract syntax trees `TyNf`, `TmNf` of "β-short, η-long" types and terms for which equality is both syntactic as well as sound and complete for judgmental equality. Given a type $\Gamma \vdash C$ type, we invert its head constructor by computing $\texttt{nfTy}(C) \in \texttt{TyNf}$, checking its head constructor in `TyNf`, and projecting its arguments.

Injectivity and invertibility are very strong conditions. As we will see in the following set of exercises, function types in set theory are not injective, nor are $\mathbf{\Pi}$-types injective in extensional type theory.

Exercise 3.5. *Give an example of three sets X, Y, Z such that $X \ncong Y$, but the set of functions $X \to Z$ is equal to the set of functions $Y \to Z$.*

Exercise 3.6. *Type theory admits a straightforward interpretation in which closed types are sets, and Exercise 3.5 shows that sets do not have injective $\mathbf{\Pi}$-types. However, this does not imply that type theory lacks injective $\mathbf{\Pi}$-types. Why not?*

Exercise 3.7. *Exhibit a context Γ and types A, B such that $\Gamma \vdash \mathbf{\Pi}(A, B) = \mathbf{\Sigma}(A, B)$ type. (Hint: you must use equality reflection and either large elimination or universes.)*

Exercise 3.8. *Exhibit a context Γ such that $\Gamma \vdash \mathbf{\Pi}(\mathbf{Unit}, \mathbf{Bool}) = \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Void})$ type. (Hint: you must use equality reflection and either large elimination or universes.)*

Warning 3.2.12. As Exercise 3.8 demonstrates, extensional type theory *does not* have injective type constructors. We will proceed under the assumption that the theory we are elaborating has invertible type constructors, and in Chapter 4 we will see how to modify our type theory to substantiate this assumption.

Completing our elaborator The force of having invertible $\mathbf{\Pi}$-types is to have an algorithm `unPi` which takes $\Gamma \vdash C$ type and returns the unique pair of types A, B for which $\Gamma \vdash C = \mathbf{\Pi}(A, B)$ type, or raises an exception if this pair does not exist. Using `unPi` we can repair our earlier attempt at checking $(\texttt{lam}\ e)$, and define the synthesis rule for $(\texttt{app}\ e_0\ e_1)$:

$$\frac{\texttt{unPi}(C) = (A, B) \qquad \Gamma.A \vdash e \Leftarrow B \rightsquigarrow b}{\Gamma \vdash (\texttt{lam}\ e) \Leftarrow C \rightsquigarrow \lambda(b)}$$

$$\frac{\Gamma \vdash e_0 \Rightarrow C \rightsquigarrow f \qquad \texttt{unPi}(C) = (A, B) \qquad \Gamma \vdash e_1 \Leftarrow A \rightsquigarrow a}{\Gamma \vdash (\texttt{app}\ e_0\ e_1) \Rightarrow B[\mathbf{id}.a] \rightsquigarrow \mathbf{app}(f, a)}$$

This is the only elaboration rule for $(\mathtt{lam}\ e)$; in particular, there is no *synthesis* rule for lambda, because we cannot elaborate e without knowing what type A to add to the context. On the other hand, to synthesize the type of $(\mathsf{app}\ e_0\ e_1)$, we *synthesize* the type of e_0; if it is of the form $\mathbf{\Pi}(A, B)$, we then *check* that e_1 has type A and then return B, suitably instantiated. Putting these rules together, the reader might notice that we cannot type-check $(\mathsf{app}\ (\mathtt{lam}\ e_0)\ e_1)$, because this would require *synthesizing* $(\mathtt{lam}\ e_0)$. In fact, bidirectional type-checking cannot type-check β-redexes in general for this reason.

For this reason, we have included a *type-annotation* preterm $(\mathsf{chk}\ e\ \tau)$ which allows users to explicitly annotate a preterm with a pretype. The type of this preterm is trivially synthesizable: it is the result of elaborating τ! In order to synthesize $(\mathsf{chk}\ e\ \tau)$, we simply *check* e against τ, and if successful, return that type.

$$\frac{\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A \qquad \Gamma \vdash e \Leftarrow A \rightsquigarrow a}{\Gamma \vdash (\mathsf{chk}\ e\ \tau) \Rightarrow A \rightsquigarrow a}$$

In particular, we can type-check the β-redex from before, as long as we annotate the lambda with its intended type: $(\mathsf{app}\ (\mathsf{chk}\ (\mathtt{lam}\ e_0)\ (\mathtt{Pi}\ \tau_0\ \tau_1))\ e_1)$.

The above rule allows us to treat a checkable term as synthesizable. The converse is much easier: to *check* the type of a synthesizable term, we simply compare the synthesized type to the expected type.

$$\frac{\Gamma \vdash e \Rightarrow B \rightsquigarrow a \qquad \Gamma \vdash A \Leftrightarrow B\,\mathsf{type}}{\Gamma \vdash e \Leftarrow A \rightsquigarrow a}$$

As written, the above rule applies to *any* checking problem because its conclusion is unconstrained. In our elaboration algorithm, we should only apply this rule if no other rule matches. It is the final “catch-all” clause for situations where we have not one but two sources of type information: on the one hand, we can synthesize e’s type directly, and on the other hand, we are also given the type that e is supposed to have. Interestingly, this is the *only* rule where our bidirectional elaborator checks type equality $\Gamma \vdash A \Leftrightarrow B\,\mathsf{type}$.

Exercise 3.9. *For each of* $(\mathtt{pair}\ e_0\ e_1)$*,* $(\mathtt{fst}\ e)$*, and* $(\mathsf{snd}\ e)$*, decide whether this preterm should be checked or synthesized, then write the algorithmic rule for elaborating it.*

A

Martin-Löf type theory

This appendix presents a substitution calculus [ML92; Tas93; Dyb96] for several variants of Martin-Löf's dependent type theory. Martin-Löf type theories are systems admitting the rules in section *Contexts and substitutions*; the rules specific to extensional type theory, those axiomatizing extensional equality types, are marked (ETT); the rules specific to intensional type theory, those axiomatizing *intensional equality types*, are marked (ITT).

Judgments

Martin-Löf type theory has four basic judgments:

1. $\vdash \Gamma\,\mathsf{cx}$ asserts that Γ is a context.

2. $\Delta \vdash \gamma : \Gamma$, presupposing $\vdash \Delta\,\mathsf{cx}$ and $\vdash \Gamma\,\mathsf{cx}$, asserts that γ is a substitution from Δ to Γ (*i.e.*, assigns a term in Δ to each variable in Γ).

3. $\Gamma \vdash A\,\mathsf{type}$, presupposing $\vdash \Gamma\,\mathsf{cx}$, asserts that A is a type in context Γ.

4. $\Gamma \vdash a : A$, presupposing $\vdash \Gamma\,\mathsf{cx}$ and $\Gamma \vdash A\,\mathsf{type}$, asserts that a is an element/term of type A in context Γ.

The *presuppositions* of a judgment are its meta-implicit-arguments, so to speak. For instance, the judgment $\Gamma \vdash A\,\mathsf{type}$ is sensible to write (is meta-well-typed) only when the judgment $\vdash \Gamma\,\mathsf{cx}$ holds. We adopt the convention that asserting the truth of a judgment implicitly asserts its well-formedness; thus asserting $\Gamma \vdash A\,\mathsf{type}$ also asserts $\vdash \Gamma\,\mathsf{cx}$.

As we assert the existence of various contexts, substitutions, types, and terms, we will simultaneously need to assert that some of these (already introduced) objects are equal to other (already introduced) objects of the same kind.

1. $\Delta \vdash \gamma = \gamma' : \Gamma$, presupposing $\Delta \vdash \gamma : \Gamma$ and $\Delta \vdash \gamma' : \Gamma$, asserts that γ, γ' are equal substitutions from Δ to Γ.

2. $\Gamma \vdash A = A'\,\mathsf{type}$, presupposing $\Gamma \vdash A\,\mathsf{type}$ and $\Gamma \vdash A'\,\mathsf{type}$, asserts that A, A' are equal types in context Γ.

3. $\Gamma \vdash a = a' : A$, presupposing $\Gamma \vdash a : A$ and $\Gamma \vdash a' : A$, asserts that a, a' are equal elements of type A in context Γ.

Two types (*resp.*, contexts, substitutions, terms) being equal has the force that it does in standard mathematics: any expression can be replaced silently by an equal expression without affecting the meaning or truth of the statement in which it appears. One important example of this principle is the "conversion rule" which states that if $\Gamma \vdash A = A'$ type and $\Gamma \vdash a : A$, then $\Gamma \vdash a : A'$.

In the rules that follow, some arguments of substitution, type, and term formers are typeset as gray subscripts; these are arguments that we will often omit because they can be inferred from context and are tedious and distracting to write.

Contexts and substitutions

$$\frac{}{\vdash \mathbf{1}\ \mathsf{cx}}\ \textsc{cx/emp} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma \vdash A\ \mathsf{type}}{\vdash \Gamma.A\ \mathsf{cx}}\ \textsc{cx/ext}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{id}_{\Gamma} : \Gamma}\ \textsc{sb/id} \qquad \frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_2 \vdash \gamma_0 \circ_{\Gamma_2,\Gamma_1,\Gamma_0} \gamma_1 : \Gamma_0}\ \textsc{sb/comp}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{id}_{\Gamma} \circ \gamma = \gamma : \Gamma} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \gamma \circ \mathbf{id}_{\Delta} = \gamma : \Gamma} \qquad \frac{\Gamma_3 \vdash \gamma_2 : \Gamma_2 \qquad \Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_3 \vdash \gamma_0 \circ (\gamma_1 \circ \gamma_2) = (\gamma_0 \circ \gamma_1) \circ \gamma_2 : \Gamma_0}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type}}{\Delta \vdash A[\gamma]_{\Delta,\Gamma}\ \mathsf{type}}\ \textsc{ty/sb} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash a[\gamma]_{\Delta,\Gamma} : A[\gamma]}\ \textsc{tm/sb}$$

$$\frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash A[\mathbf{id}_{\Gamma}] = A\ \mathsf{type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash a[\mathbf{id}_{\Gamma}] = a : A}$$

$$\frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash A\ \mathsf{type}}{\Gamma_2 \vdash A[\gamma_0 \circ \gamma_1] = A[\gamma_0][\gamma_1]\ \mathsf{type}} \qquad \frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash a : A}{\Gamma_2 \vdash a[\gamma_0 \circ \gamma_1] = a[\gamma_0][\gamma_1] : A[\gamma_0 \circ \gamma_1]}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash !_{\Gamma} : \mathbf{1}}\ \textsc{sb/emp} \qquad \frac{\Gamma \vdash \delta : \mathbf{1}}{\Gamma \vdash !_{\Gamma} = \delta : \mathbf{1}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \gamma ._{\Delta,\Gamma,A} a : \Gamma.A}\ \textsc{sb/ext} \qquad \frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma.A \vdash \mathbf{p}_{\Gamma,A} : \Gamma}\ \textsc{sb/wk}$$

$$\frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma.A \vdash \mathbf{q}_{\Gamma,A} : A[\mathbf{p}_{\Gamma,A}]}\ \textsc{var} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{p}_{\Gamma,A} \circ_{\Gamma.A} (\gamma.a) = \gamma : \Gamma} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{q}_{\Gamma,A}[\gamma.a] = a : A[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma.A}{\Delta \vdash \gamma = (\mathbf{p}_{\Gamma,A} \circ_{\Gamma.A} \gamma).(\mathbf{q}_{\Gamma,A}[\gamma]) : \Gamma.A}$$

Π-*types*

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type}}{\Gamma \vdash \Pi_\Gamma(A,B)\ \mathsf{type}}\ \textsc{pi/form} \qquad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \lambda_{\Gamma,A,B}(b) : \Pi(A,B)}\ \textsc{pi/intro}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash f : \Pi(A,B)}{\Gamma \vdash \mathbf{app}_{\Gamma,A,B}(f,a) : B[\mathbf{id}_\Gamma.a]}\ \textsc{pi/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type}}{\Delta \vdash \Pi_\Gamma(A,B)[\gamma] = \Pi_\Delta(A[\gamma], B[(\gamma \circ \mathbf{p}_{\Delta,A[\gamma]}).\mathbf{q}_{\Delta,A[\gamma]}])\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash b : B}{\Delta \vdash \lambda(b)[\gamma] = \lambda(b[(\gamma \circ \mathbf{p}).\mathbf{q}]) : \Pi(A,B)[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash f : \Pi(A,B)}{\Delta \vdash \mathbf{app}(f,a)[\gamma] = \mathbf{app}(f[\gamma], a[\gamma]) : B[(\mathbf{id}_\Gamma.a) \circ \gamma]}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \mathbf{app}(\lambda(b), a) = b[\mathbf{id}.a] : B[\mathbf{id}.a]} \qquad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash f : \Pi(A,B)}{\Gamma \vdash f = \lambda(\mathbf{app}(f[\mathbf{p}_{\Gamma,A}], \mathbf{q}_{\Gamma,A})) : \Pi(A,B)}$$

Σ-*types*

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type}}{\Gamma \vdash \Sigma_\Gamma(A,B)\ \mathsf{type}}\ \textsc{sigma/form}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}_\Gamma.a]}{\Gamma \vdash \mathbf{pair}_{\Gamma,A,B}(a,b) : \Sigma(A,B)}\ \textsc{sigma/intro}$$

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{fst}_{\Gamma,A,B}(p) : A}\ \textsc{sigma/elim/fst}$$

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{snd}_{\Gamma,A,B}(p) : B[\mathbf{id}_\Gamma.\mathbf{fst}(p)]}\ \textsc{sigma/elim/snd}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type}}{\Delta \vdash \Sigma_\Gamma(A,B)[\gamma] = \Sigma_\Delta(A[\gamma], B[(\gamma \circ \mathbf{p}).\mathbf{q}])\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{pair}(a,b)[\gamma] = \mathbf{pair}(a[\gamma], b[\gamma]) : \Sigma(A,B)[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{fst}(p)[\gamma] = \mathbf{fst}(p[\gamma]) : A[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{snd}(p)[\gamma] = \mathbf{snd}(p[\gamma]) : B[(\mathbf{id}.\mathbf{fst}(p)) \circ \gamma]}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{fst}(\mathbf{pair}(a,b)) = a : A}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{snd}(\mathbf{pair}(a,b)) = b : B[\mathbf{id}.a]}$$

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma.A \vdash B\ \mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash p = \mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p)) : \Sigma(A,B)}$$

Extensional equality types

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \mathbf{Eq}_\Gamma(A,a,b)\ \mathsf{type}}\ \textsc{eq/form (ett)} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl}_{\Gamma,A,a} : \mathbf{Eq}(A,a,a)}\ \textsc{eq/intro (ett)}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Delta \vdash \mathbf{Eq}_\Gamma(A,a,b)[\gamma] = \mathbf{Eq}_\Delta(A[\gamma], a[\gamma], b[\gamma])\ \mathsf{type}}\ \text{(ETT)}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash \mathbf{refl}[\gamma] = \mathbf{refl} : \mathbf{Eq}(A,a,a)[\gamma]}\ \text{(ETT)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash a = b : A}\ (\text{ETT})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash p = \mathbf{refl} : \mathbf{Eq}(A, a, b)}\ (\text{ETT})$$

Unit type

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Unit}_\Gamma\ \mathsf{type}}\ \textsc{unit/form} \qquad \frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{tt}_\Gamma : \mathbf{Unit}}\ \textsc{unit/intro}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Unit}_\Gamma[\gamma] = \mathbf{Unit}_\Delta\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{tt}_\Gamma[\gamma] = \mathbf{tt}_\Delta : \mathbf{Unit}} \qquad \frac{\Gamma \vdash a : \mathbf{Unit}}{\Gamma \vdash a = \mathbf{tt} : \mathbf{Unit}}$$

Empty type

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Void}_\Gamma\ \mathsf{type}}\ \textsc{empty/form} \qquad \frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathsf{type}}{\Gamma \vdash \mathbf{absurd}_{\Gamma,A}(b) : A[\mathbf{id}.b]}\ \textsc{empty/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Void}_\Gamma[\gamma] = \mathbf{Void}_\Delta\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathsf{type}}{\Delta \vdash \mathbf{absurd}(b)[\gamma] = \mathbf{absurd}(b[\gamma]) : A[\gamma.b[\gamma]]}$$

Boolean type

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Bool}_\Gamma\ \mathsf{type}}\ \textsc{bool/form}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{true}_\Gamma : \mathbf{Bool}}\ \textsc{bool/intro/true} \qquad \frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{false}_\Gamma : \mathbf{Bool}}\ \textsc{bool/intro/false}$$

$$\frac{\begin{array}{c}\Gamma \vdash b : \mathbf{Bool} \\ \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]\end{array}}{\Gamma \vdash \mathbf{if}_{\Gamma,A}(a_t, a_f, b) : A[\mathbf{id}.b]}\ \textsc{bool/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Bool}_{\Gamma}[\gamma] = \mathbf{Bool}_{\Delta}\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{true}_{\Gamma}[\gamma] = \mathbf{true}_{\Delta} : \mathbf{Bool}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{false}_{\Gamma}[\gamma] = \mathbf{false}_{\Delta} : \mathbf{Bool}}$$

$$\frac{\begin{array}{c}\Delta \vdash \gamma : \Gamma \\ \Gamma \vdash b : \mathbf{Bool} \qquad \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]\end{array}}{\Delta \vdash \mathbf{if}(a_t, a_f, b)[\gamma] = \mathbf{if}(a_t[\gamma], a_f[\gamma], b[\gamma]) : A[\gamma.b[\gamma]]}$$

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}(a_t, a_f, \mathbf{true}) = a_t : A[\mathbf{id}.\mathbf{true}]}$$

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}(a_t, a_f, \mathbf{false}) = a_f : A[\mathbf{id}.\mathbf{false}]}$$

Natural number type

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Nat}_{\Gamma}\ \mathsf{type}}\ \textsc{nat/form}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{zero}_{\Gamma} : \mathbf{Nat}}\ \textsc{nat/intro/zero} \qquad \frac{\Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{suc}_{\Gamma}(n) : \mathbf{Nat}}\ \textsc{nat/intro/suc}$$

$$\frac{\begin{array}{c}\Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \qquad \Gamma \vdash n : \mathbf{Nat}\end{array}}{\Gamma \vdash \mathbf{rec}_{\Gamma,A}(a_z, a_s, n) : A[\mathbf{id}.n]}\ \textsc{nat/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Nat}_{\Gamma}[\gamma] = \mathbf{Nat}_{\Delta}\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{zero}_{\Gamma}[\gamma] = \mathbf{zero}_{\Delta} : \mathbf{Nat}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash n : \mathbf{Nat}}{\Delta \vdash \mathbf{suc}_{\Gamma}(n)[\gamma] = \mathbf{suc}_{\Delta}(n[\gamma]) : \mathbf{Nat}}$$

$$\frac{\begin{array}{c}\Delta \vdash \gamma : \Gamma \qquad \Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \qquad \Gamma \vdash n : \mathbf{Nat}\end{array}}{\Delta \vdash \mathbf{rec}(a_z, a_s, n)[\gamma] = \mathbf{rec}(a_z[\gamma], a_s[(\gamma \circ \mathbf{p}^2).\mathbf{q}[\mathbf{p}].\mathbf{q}], n[\gamma]) : A[\gamma.n[\gamma]]}$$

$$\frac{\Gamma.\mathbf{Nat} \vdash A \,\mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{zero}) = a_z : A[\mathbf{id}.\mathbf{zero}]}$$

$$\frac{\begin{array}{c}\Gamma.\mathbf{Nat} \vdash A \,\mathsf{type} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \qquad \Gamma \vdash n : \mathbf{Nat}\end{array}}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{suc}(n)) = a_s[\mathbf{id}.n.\mathbf{rec}(a_z, a_s, n)] : A[\mathbf{id}.\mathbf{suc}(n)]}$$

Intensional equality types

$$\frac{\Gamma \vdash a, b : A}{\Gamma \vdash \mathbf{Id}_\Gamma(A, a, b) \,\mathsf{type}}\ \textsc{id/form (itt)} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl}_{\Gamma,A,a} : \mathbf{Id}(A, a, a)}\ \textsc{id/intro (itt)}$$

$$\frac{\begin{array}{c}\Gamma \vdash a_0, a_1 : A \qquad \Gamma \vdash p : \mathbf{Id}(A, a_0, a_1) \\ \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q}) \vdash B \,\mathsf{type} \qquad \Gamma.A \vdash b : B[\mathbf{p}.\mathbf{q}.\mathbf{q}.\mathbf{refl}]\end{array}}{\Gamma \vdash \mathbf{J}_{\Gamma,A,a_0,a_1,B}(b, p) : B[\mathbf{id}.a_0.a_1.p]}\ \textsc{id/elim (itt)}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a, b : A}{\Delta \vdash \mathbf{Id}_\Gamma(A, a, b)[\gamma] = \mathbf{Id}_\Delta(A[\gamma], a[\gamma], b[\gamma]) \,\mathsf{type}}\ \textsc{(itt)}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash \mathbf{refl}[\gamma] = \mathbf{refl} : \mathbf{Id}(A[\gamma], a[\gamma], a[\gamma])}\ \textsc{(itt)}$$

$$\frac{\begin{array}{c}\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a_0, a_1 : A \qquad \Gamma \vdash p : \mathbf{Id}(A, a_0, a_1) \\ \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q}) \vdash B \,\mathsf{type} \qquad \Gamma.A \vdash b : B[\mathbf{p}.\mathbf{q}.\mathbf{q}.\mathbf{refl}]\end{array}}{\Delta \vdash \mathbf{J}(b, p)[\gamma] = \mathbf{J}(b[(\gamma \circ \mathbf{p}).\mathbf{q}], p[\gamma]) : B[\gamma.a_0[\gamma].a_1[\gamma].p[\gamma]]}\ \textsc{(itt)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q}) \vdash B \,\mathsf{type} \qquad \Gamma.A \vdash b : B[\mathbf{p}.\mathbf{q}.\mathbf{q}.\mathbf{refl}]}{\Gamma \vdash \mathbf{J}(b, \mathbf{refl}) = b[\mathbf{id}.a.a.\mathbf{refl}] : B[\mathbf{id}.a.a.\mathbf{refl}]}\ \textsc{(itt)}$$

Universes

$$\frac{\vdash \Gamma \,\mathsf{cx}}{\Gamma \vdash \mathbf{U}_{\Gamma,i} \,\mathsf{type}}\ \textsc{uni/form} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_{i,\Gamma}(a) \,\mathsf{type}}\ \textsc{el/form}$$

$$\frac{\Gamma \vdash c_0 : \mathbf{U}_i \qquad \Gamma.\mathbf{El}(c_0) \vdash c_1 : \mathbf{U}_i}{\Gamma \vdash \mathbf{pi}_{i,\Gamma}(c_0, c_1) : \mathbf{U}_i}\ \textsc{pi/code} \qquad \frac{\Gamma \vdash c_0 : \mathbf{U}_i \qquad \Gamma.\mathbf{El}(c_0) \vdash c_1 : \mathbf{U}_i}{\Gamma \vdash \mathbf{sig}_{i,\Gamma}(c_0, c_1) : \mathbf{U}_i}\ \textsc{sig/code}$$

$$\frac{\Gamma \vdash c : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}(c)}{\Gamma \vdash \mathbf{eq}_{i,\Gamma}(c, x, y) : \mathbf{U}_i}\ \textsc{eq/code (ett)} \qquad \frac{\Gamma \vdash c : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}(c)}{\Gamma \vdash \mathbf{id}_{i,\Gamma}(c, x, y) : \mathbf{U}_i}\ \textsc{id/code (itt)}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{unit}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{unit/code} \qquad \frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{nat}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{nat/code} \qquad \frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{void}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{empty/code}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{bool}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{bool/code} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad j < i}{\Gamma \vdash \mathbf{uni}_{\Gamma,i,j} : \mathbf{U}_i}\ \textsc{uni/code} \qquad \frac{\Gamma \vdash c : \mathbf{U}_i}{\Gamma \vdash \mathbf{lift}_{i,\Gamma}(c) : \mathbf{U}_{i+1}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{U}_{\Gamma,i}[\gamma] = \mathbf{U}_{\Delta,i}\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i}{\Delta \vdash \mathbf{El}(a)[\gamma] = \mathbf{El}(a[\gamma])\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash c_0 : \mathbf{U}_i \qquad \Gamma.\mathbf{El}(c_0) \vdash c_1 : \mathbf{U}_i}{\Delta \vdash \mathbf{pi}(c_0, c_1)[\gamma] = \mathbf{pi}(c_0[\gamma], c_1[(\gamma \circ \mathbf{p}).\mathbf{q}]) : \mathbf{U}_i}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash c_0 : \mathbf{U}_i \qquad \Gamma.\mathbf{El}(c_0) \vdash c_1 : \mathbf{U}_i}{\Delta \vdash \mathbf{sig}(c_0, c_1)[\gamma] = \mathbf{sig}(c_0[\gamma], c_1[(\gamma \circ \mathbf{p}).\mathbf{q}]) : \mathbf{U}_i}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash c : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}(c)}{\Delta \vdash \mathbf{eq}(c, x, y)[\gamma] = \mathbf{eq}(c[\gamma], x[\gamma], y[\gamma]) : \mathbf{U}_i}\ \text{(ETT)}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash c : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}(c)}{\Delta \vdash \mathbf{id}(c, x, y)[\gamma] = \mathbf{id}(c[\gamma], x[\gamma], y[\gamma]) : \mathbf{U}_i}\ \text{(ITT)}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{unit}[\gamma] = \mathbf{unit} : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{nat}[\gamma] = \mathbf{nat} : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{void}[\gamma] = \mathbf{void} : \mathbf{U}_i}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{bool}[\gamma] = \mathbf{bool} : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad j < i}{\Delta \vdash \mathbf{uni}_j[\gamma] = \mathbf{uni}_j : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash c : \mathbf{U}_i}{\Delta \vdash \mathbf{lift}(c)[\gamma] = \mathbf{lift}(c[\gamma]) : \mathbf{U}_{i+1}}$$

$$\frac{\Gamma \vdash c_0 : \mathbf{U}_i \qquad \Gamma.\mathbf{El}(c_0) \vdash c_1 : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}(\mathbf{pi}(c_0, c_1)) = \Pi(\mathbf{El}(c_0), \mathbf{El}(c_1))\ \mathsf{type}}$$

$$\frac{\Gamma \vdash c_0 : \mathbf{U}_i \qquad \Gamma.\mathbf{El}(c_0) \vdash c_1 : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}(\mathbf{sig}(c_0, c_1)) = \Sigma(\mathbf{El}(c_0), \mathbf{El}(c_1))\ \mathsf{type}}$$

$$\frac{\Gamma \vdash c : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}(c)}{\Gamma \vdash \mathbf{El}(\mathbf{eq}(c, x, y)) = \mathbf{Eq}(\mathbf{El}(c), x, y) \ \mathsf{type}} \ \text{(ETT)}$$

$$\frac{\Gamma \vdash c : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}(c)}{\Gamma \vdash \mathbf{El}(\mathbf{id}(c, x, y)) = \mathbf{Id}(\mathbf{El}(c), x, y) \ \mathsf{type}} \ \text{(ITT)} \qquad \frac{}{\Gamma \vdash \mathbf{El}(\mathbf{unit}) = \mathbf{Unit} \ \mathsf{type}}$$

$$\frac{}{\Gamma \vdash \mathbf{El}(\mathbf{nat}) = \mathbf{Nat} \ \mathsf{type}} \qquad \frac{}{\Gamma \vdash \mathbf{El}(\mathbf{void}) = \mathbf{Void} \ \mathsf{type}} \qquad \frac{}{\Gamma \vdash \mathbf{El}(\mathbf{bool}) = \mathbf{Bool} \ \mathsf{type}}$$

$$\frac{j < i}{\Gamma \vdash \mathbf{El}(\mathbf{uni}_j) = \mathbf{U}_j \ \mathsf{type}} \qquad \frac{\Gamma \vdash c : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_{i+1}(\mathbf{lift}(c)) = \mathbf{El}_i(c) \ \mathsf{type}}$$

Bibliography

[Abe13] Andreas Abel. "Normalization by Evaluation: Dependent Types and Impredicativity". Habilitation thesis. Ludwig-Maximilians-Universität München, 2013. URL: http://www2.tcs.ifi.lmu.de/~abel/habil.pdf.

[ACD07] Andreas Abel, Thierry Coquand, and Peter Dybjer. "Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements". In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. July 2007, pp. 3–12. DOI: 10.1109/LICS.2007.33.

[Agda] The Agda Development Team. *The Agda Programming Language*. 2020. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

[AK16] Thorsten Altenkirch and Ambrus Kaposi. "Type theory in type theory using quotient inductive types". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, Jan. 2016. DOI: 10.1145/2837614.2837638. URL: http://dx.doi.org/10.1145/2837614.2837638.

[Alt+01] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. "Normalization by evaluation for typed lambda calculus with coproducts". In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 2001, pp. 303–310. DOI: 10.1109/LICS.2001.932506.

[Alt23] Thorsten Altenkirch. "Should Type Theory Replace Set Theory as the Foundation of Mathematics?" In: *Global Philosophy* 33.21 (2023). DOI: 10.1007/s10516-023-09676-0.

[AMB13] Guillaume Allais, Conor McBride, and Pierre Boutillier. "New Equations for Neutral Terms: A Sound and Complete Decision Procedure, Formalized". In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-Typed Programming*. DTP '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 13–24. ISBN: 9781450323840. DOI: 10.1145/2502409.2502411. URL: https://doi.org/10.1145/2502409.2502411.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational Equality, Now!" In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. PLPV '07. New York, NY, USA: ACM, 2007, pp. 57–68. ISBN: 978-1-59593-677-6. DOI: 10.1145/1292597.1292608.

[Ang+21] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. "Syntax and models of Cartesian cubical type theory". In: *Mathematical Structures in Computer Science* 31.4 (2021). Special issue on Homotopy Type Theory and Univalent Foundations, pp. 424–468. DOI: 10.1017/S0960129521000347.

[Aug99] Lennart Augustsson. "Cayenne — A Language with Dependent Types". In: *Advanced Functional Programming*. Ed. by S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 240–267. ISBN: 978-3-540-48506-3. DOI: 10.1007/10704973_6.

[AW09] Steve Awodey and Michael A. Warren. "Homotopy theoretic models of identity types". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 146.1 (Jan. 2009), pp. 45–55. ISSN: 0305-0041. DOI: 10.1017/S0305004108001783.

[Awo18] Steve Awodey. "Natural models of homotopy type theory". In: *Mathematical Structures in Computer Science* 28.2 (2018), pp. 241–286. DOI: 10.1017/S0960129516000268.

[Bar91] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of Functional Programming* 1.2 (1991), pp. 125–154. DOI: 10.1017/S0956796800020025.

[Bau+24] Andrej Bauer, Gaëtan Gilbert, Philipp Haselwarter, Anja Petković, Matija Pretnar, and Chris Stone. *Andromeda*. Github. 2024. URL: https://github.com/Andromedans/andromeda.

[Bra13] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X.

[Bra17] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications, 2017. ISBN: 9781617293023.

[Bru72] N. G. de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0.

[Car86] John Cartmell. "Generalised algebraic theories and contextual categories". In: *Annals of Pure and Applied Logic* 32 (1986), pp. 209–243. ISSN: 0168-0072. DOI: 10.1016/0168-0072(86)90053-9.

[CCD20] Simon Castellan, Pierre Clairambault, and Peter Dybjer. *Categories with Families: Unityped, Simply Typed, and Dependently Typed*. 2020. arXiv: 1904.00827 [cs.LO].

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. “Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom”. In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by Tarmo Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 5:1–5:34. ISBN: 978-3-95977-030-9. DOI: 10.4230/LIPIcs.TYPES.2015.5.

[CH88] Thierry Coquand and Gérard Huet. “The Calculus of Constructions”. In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3.

[Chr23] David Thrane Christiansen. *Functional Programming in Lean*. 2023. URL: https://lean-lang.org/functional_programming_in_lean/.

[Con+85] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development Environment*. Prentice-Hall, 1985. URL: http://www.nuprl.org/book/.

[Coq] The Coq Development Team. *The Coq Proof Assistant*. 2020. URL: https://www.coq.inria.fr.

[Coq13] Thierry Coquand. “Presheaf model of type theory”. Unpublished note. 2013. URL: http://www.cse.chalmers.se/~coquand/presheaf.pdf.

[Coq19] Thierry Coquand. “Canonicity and normalization for dependent type theory”. In: *Theoretical Computer Science* 777 (2019). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I, pp. 184–191. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.01.015.

[Coq86] Thierry Coquand. “An Analysis of Girard’s Paradox”. In: *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*. Cambridge, MA, USA: IEEE Computer Society Press, 1986, pp. 227–236.

[Coq96] Thierry Coquand. “An algorithm for type-checking dependent types”. In: *Science of Computer Programming* 26.1 (1996), pp. 167–177. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(95)00021-6.

[CP90] Thierry Coquand and Christine Paulin. “Inductively defined types”. In: *COLOG-88*. Ed. by Per Martin-Löf and Grigori Mints. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 50–66. ISBN: 978-3-540-46963-6. DOI: 10.1007/3-540-52335-9_47.

[Cro94] Roy L. Crole. *Categories for Types*. Cambridge: Cambridge University Press, 1994. DOI: 10.1017/CBO9781139172707.

[Dow93] Gilles Dowek. "The undecidability of typability in the Lambda-Pi-calculus". In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–145. ISBN: 978-3-540-47586-6.

[Dyb00] Peter Dybjer. "A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory". In: *The Journal of Symbolic Logic* 65.2 (2000), pp. 525–549. ISSN: 00224812. DOI: 10.2307/2586554.

[Dyb94] Peter Dybjer. "Inductive families". In: *Formal Aspects of Computing* 6.4 (July 1994), pp. 440–465. ISSN: 0934-5043. DOI: 10.1007/BF01211308.

[Dyb96] Peter Dybjer. "Internal type theory". In: *Types for Proofs and Programs (TYPES 1995)*. Ed. by Stefano Berardi and Mario Coppo. Vol. 1158. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134. ISBN: 978-3-540-70722-6. DOI: 10.1007/3-540-61780-9_66.

[FAM23] Kuen-Bang Hou (Favonia), Carlo Angiuli, and Reed Mullanix. "An Order-Theoretic Analysis of Universe Polymorphism". In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023). DOI: 10.1145/3571250.

[FC18] Daniel P. Friedman and David Thrane Christiansen. *The Little Typer*. The MIT Press, 2018. ISBN: 9780262536431.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer Berlin Heidelberg, 1979. ISBN: 9783540385264. DOI: 10.1007/3-540-09724-4.

[Gra09] Johan Georg Granström. "Reference and Computation in Intuitionistic Type Theory". PhD thesis. Uppsala University, 2009. URL: https://intuitionistic.files.wordpress.com/2010/07/theses_published_uppsala.pdf.

[Gra+21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. "Multimodal Dependent Type Theory". In: *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). DOI: 10.46298/lmcs-17(3:11)2021.

[Gra+22] Daniel Gratzer, Evan Cavallo, G. A. Kavvos, Adrien Guatto, and Lars Birkedal. "Modalities and Parametric Adjoints". In: *ACM Transactions on Computational Logic* 23.3 (Apr. 2022). ISSN: 1529-3785. DOI: 10.1145/3514241.

[Gra23] Daniel Gratzer. "Syntax and semantics of modal type theory". PhD thesis. Aarhus University, 2023. URL: https://pure.au.dk/portal/en/publications/syntax-and-semantics-of-modal-type-theory(694f77d2-47d3-4986-bb82-129b8d96206e).html.

[Har09] John Harrison. “HOL Light: An Overview”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 60–66. ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_4.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[Has21] Philipp Haselwarter. “Effective Metatheory for Type Theory”. PhD thesis. University of Ljubljana, 2021.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. “A framework for defining logics”. In: *J. ACM* 40.1 (Jan. 1993), 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060.

[HM95] Robert Harper and Greg Morrisett. “Compiling Polymorphism Using Intensional Type Analysis”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’95. New York, NY, USA: ACM, 1995, pp. 130–141. ISBN: 0897916921. DOI: 10.1145/199448.199475.

[Hof95] Martin Hofmann. “On the interpretation of type theory in locally cartesian closed categories”. In: *8th Workshop, Computer Science Logic (CSL 1994)*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Vol. 933. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 427–441. ISBN: 978-3-540-49404-1. DOI: 10.1007/BFb0022273.

[Hof97] Martin Hofmann. “Syntax and Semantics of Dependent Types”. In: *Semantics and Logics of Computation*. Ed. by Andrew M. Pitts and P. Dybjer. Publications of the Newton Institute. Cambridge University Press, 1997, pp. 79–130. DOI: 10.1017/CBO9780511526619.004.

[Hur95] Antonius J. C. Hurkens. “A simplification of Girard’s paradox”. In: *Typed Lambda Calculi and Applications*. Ed. by Mariangiola Dezani-Ciancaglini and Gordon Plotkin. TLCA 1995. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 266–278. ISBN: 978-3-540-49178-1. DOI: 10.1007/BFb0014058.

[KKA19] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. “Constructing quotient inductive-inductive types”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019). DOI: 10.1145/3290315.

[Kov22] András Kovács. “Type-Theoretic Signatures for Algebraic Theories and Inductive Types”. PhD thesis. Eötvös Loránd University, 2022.

[LW15] Peter LeFanu Lumsdaine and Michael A. Warren. "The Local Universes Model: An Overlooked Coherence Construction for Dependent Type Theories". In: *ACM Transactions on Computational Logic* 16.3 (2015). DOI: 10.1145/2754931.

[McB18] Conor McBride. *Basics of bidirectionalism.* Online. 2018. URL: https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/.

[McB19] Conor McBride. *The types who say 'ni'.* Online. 2019. URL: https://github.com/pigworker/TypesWhoSayNi/blob/master/tex/TypesWhoSayNi.pdf.

[Mim20] Samuel Mimram. *PROGRAM = PROOF.* Independently published, 2020. ISBN: 979-8615591839. URL: http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf.

[ML71] Per Martin-Löf. "An intuitionistic theory of types". Unpublished preprint. 1971.

[ML75] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73.* Ed. by H. E. Rose and J. C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. North-Holland, 1975, pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1.

[ML82] Per Martin-Löf. "Constructive mathematics and computer programming". In: *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979.* Ed. by L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski. Vol. 104. Studies in Logic and the Foundations of Mathematics. North-Holland, 1982, pp. 153–175. DOI: 10.1016/S0049-237X(09)70189-2.

[ML84] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.* Vol. 1. Studies in Proof Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.

[ML87] Per Martin-Löf. "Truth of a Proposition, Evidence of a Judgement, Validity of a Proof". In: *Synthese* 73.3 (1987), pp. 407–420. DOI: 10.1007/bf00484985.

[ML92] Per Martin-Löf. *Substitution calculus.* Notes from a lecture given in Göteborg. 1992.

[ML96] Per Martin-Löf. "On the meanings of the logical constants and the justifications of the logical laws". In: *Nordic journal of philosophical logic* 1.1 (1996), pp. 11–60.

[MU21] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5. DOI: 10.1007/978-3-030-79876-5_37.

[NFS12] Fredrik Nordvall Forsberg and Anton Setzer. "A finite axiomatisation of inductive-inductive definitions". In: *Logic, Construction, Computation*. Ed. by Ulrich Berger, Diener Hannes, Peter Schuster, and Monika Seisenberger. Vol. 3. Ontos mathematical logic. Ontos Verlag, 2012, pp. 259–287. DOI: 10.1515/9783110324921.259.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990. URL: http://www.cse.chalmers.se/research/group/logic/book/.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. DOI: 10.1007/3-540-45949-9.

[PD01] Frank Pfenning and Rowan Davies. "A judgmental reconstruction of modal logic". In: *Mathematical Structures in Computer Science* 11.4 (2001), pp. 511–540. DOI: 10.1017/S0960129501003322.

[Pol02] Robert Pollack. "Dependently Typed Records in Type Theory". In: *Formal Aspects of Computing* 13.3–5 (July 2002), pp. 386–402. ISSN: 0934-5043. DOI: 10.1007/s001650200018.

[PT00] Benjamin C. Pierce and David N. Turner. "Local type inference". In: *ACM Transactions Programming Language and Systems* 22.1 (2000), pp. 1–44.

[PT22] Loïc Pujet and Nicolas Tabareau. "Observational Equality: Now for Good". In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). DOI: 10.1145/3498693.

[SAG22] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. "A Cubical Language for Bishop Sets". In: *Logical Methods in Computer Science* 18 (1 Mar. 2022). DOI: 10.46298/lmcs-18(1:43)2022.

[Shu19] Michael Shulman. *All $(\infty, 1)$-toposes have strict univalent universes*. Preprint. Apr. 2019. arXiv: 1904.07004 [math.AT].

[Shu21] Michael Shulman. "Homotopy Type Theory: The Logic of Space". In: *New Spaces in Mathematics: Formal and Conceptual Reflections*. Ed. by Mathieu Anel and Gabriel Catren. Vol. 1. Cambridge University Press, 2021. Chap. 6, pp. 322–404. DOI: 10.1017/9781108854429.009.

[Smi88] Jan M. Smith. “The Independence of Peano’s Fourth Axiom from Martin-Löf’s Type Theory Without Universes”. In: *The Journal of Symbolic Logic* 53.3 (1988), pp. 840–845. ISSN: 00224812. DOI: 10.2307/2274575.

[Ste19] Jonathan Sterling. *Algebraic Type Theory and Universe Hierarchies*. 2019. arXiv: 1902.08848 [cs.LO].

[Stu16] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, 2016. ISBN: 9781970001273. DOI: 10.1145/2841316.

[Tas93] Álvaro Tasistro. *Formulation of Martin-Löf’s theory of types with explicit substitutions*. Licentiate thesis, Chalmers University of Technology and University of Göteborg. 1993.

[Tse17] Dimitris Tsementzis. “Univalent foundations as structuralist foundations”. In: *Synthese* 194.9 (2017), pp. 3583–3617. DOI: 10.1007/s11229-016-1109-x.

[Uem21] Taichi Uemura. “Abstract and Concrete Type Theories”. PhD thesis. Institute for Logic, Language and Computation, University of Amsterdam, 2021.

[UF13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Self-published, 2013. URL: https://homotopytypetheory.org/book/.

[Vaz+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. “Refinement Types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 269–282. ISBN: 9781450328739. DOI: 10.1145/2628136.2628161.

[WKS22] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. Aug. 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

[Xi07] Hongwei Xi. “Dependent ML: An approach to practical programming with dependent types”. In: *Journal of Functional Programming* 17.2 (2007), pp. 215–286. DOI: 10.1017/S0956796806006216.