Principles and Pragmatics of Dependent Types

Carlo Angiuli
Indiana University

Daniel Gratzer
Aarhus University

(1/15/2024)

Acknowledgements

We thank Lars Birkedal for his comments and suggestions on drafts of these notes.

Contents

Acknowledgements **ii**

Contents **iii**

1 Introduction **1**
- 1.1 Dependent types for functional programmers 2

2 Extensional type theory **14**
- 2.1 The simply-typed lambda calculus 15
- 2.2 Towards the syntax of dependent type theory 18
- 2.3 The well-behaved fragment: Π, Σ, **Eq**, **Unit** 25
- 2.4 Inductive types in Martin-Löf type theory 35
- 2.5 Large elimination and universes 45

3 Intensional type theory **56**
- 3.1 The anatomy of a bare-bones proof assistant 57
- 3.2 The theory behind a bare-bones proof assistant 58
- 3.3 Extensional equality yields undecidable type-checking 58
- 3.4 Internalizing equality in type theory 59
- 3.5 Universes as a case study in elaboration 59

4 Vistas **60**

A Martin-Löf type theory **61**

Bibliography **65**

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 $A \longrightarrow B$ 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."

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

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:

```
head : (A : Set) → List A → A
head A [] = error "List must be non-empty."
head A (x :: xs) = x

tail : (A : Set) → List A → List A
tail A [] = error "List must be non-empty."
tail A (x :: xs) = xs
```

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 $1 + ($head **Nat** $(1 :: []))$ has type **Nat** whereas $1 + ($head **String** $($`"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]

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A 0
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

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

Now the types of concrete lists are more informative—$(2 :: 1 :: []) :$ **Vec Int** 2 and $(1 :: []) :$ **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.

```
head : {A : Set} {n : Nat} → Vec A (suc n) → A
-- head [] is impossible
head (x ∷ xs) = x

tail : {A : Set} {n : Nat} → Vec A (suc n) → Vec A n
-- tail [] is impossible
tail (x ∷ xs) = xs
```

Consider now the operation that concatenates two vectors:

```
append : {A : Set} {n : Nat} {m : Nat} → Vec A n → Vec A m → 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?

```
filter : {A : Set} {n : Nat} → (A → Bool) → Vec A n → Vec A ?
```

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:

```
filter : {A : Set} {n : Nat} → (f : A → Bool) → (l : Vec A n) → Vec A (filterLen f l)

filterLen : {A : Set} {n : Nat} → (A → Bool) → Vec A n → Nat
filterLen f [] = 0
filterLen f (x ∷ xs) = if f(x) then suc (filterLen f xs) else filterLen f xs
```

As before, once f and l are known the type of filter $f\ l$: **Vec** A (filterLen $f\ l$) will simplify by evaluating filterLen $f\ l$, but as long as either remains a variable we cannot learn much by computation. Nevertheless, filterLen has many properties of interest: filterLen $f\ l$ is at most the length of l, filterLen $(\lambda x \longrightarrow \text{false})\ l$ 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 **Vec** $A\ n$ with **List** A before applying a standard non-dependent type-checking algorithm. This is possible because we can regard the dependency in **Vec** $A\ n$ as expressing a computable *refinement*—or subset—of the non-dependent type of lists, namely $\{l : \textbf{List}\ A \mid \text{length}\ l = n\}$.

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:

$$\text{filter} : \textbf{Vec}\ A\ m \longrightarrow (\{n : \textbf{Nat} \mid n \leq m\} \times \textbf{Vec}\ A\ n)$$

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 *non-uniformly term-indexed* families of types, as in the following example:

```
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:

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

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*, $\mathsf{args}\ (\mathsf{natTok} :: \mathsf{toks})$ and $\mathbf{Nat} \to \mathsf{args}\ \mathsf{toks}$ 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. ⋄

add forward ref to Ch 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 **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 **Vec** A (filterLen ($\lambda x \to$ false) l) rather than **Vec** A 0 as desired, and in this case the expression filterLen ($\lambda x \to$ 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 : **Vec** A n, either:

- $l = []$, in which case filterLen ($\lambda x \to$ 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 ($\lambda x \to$ false) $xs = 0$ and thus filterLen ($\lambda x \to$ 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 ($\lambda x \to$ false) $l \equiv 0$:

```
_≡_ : {A : Set} → A → A → Set
refl : {A : Set} {x : A} → x ≡ x

lemma : {A : Set} {n : Nat} → (l : Vec A n) → filterLen (λl → false) l ≡ 0
lemma [] = refl
lemma (x :: xs) = lemma xs
```

The [] clause of lemma ought to have type filterLen ($\lambda l \to$ 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 ($\lambda l \to$ false) $(x :: xs) \equiv 0$, which is definitionally equal to filterLen ($\lambda l \to$ false) $xs \equiv 0$, the expected type of the recursive call lemma xs.

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

```
subst : {A : Set} {x y : A} → (P : A → Set) → x ≡ y → P(x) → P(y)

filterAll : {A : Set} {n : Nat} → Vec A n → Vec A 0
filterAll {A} l = subst (Vec A) (lemma l) (filter (λx → false) l)
```

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 : **String**, the final return type of sprintf s is **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 3 and 4.

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 ($\lambda x \rightarrow$ false) ls). At this point, filter ($\lambda x \rightarrow$ false) ls already has the desired type **Vec** A 0 because filterLen ($\lambda x \rightarrow$ 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].

A Martin-Löf type theory

This appendix presents a substitution calculus [ML92; Tas93; Dyb96] for Martin-Löf's extensional dependent type theory.

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 $\Gamma \vdash A = A'\ \mathsf{type}$ 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*

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

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

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

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

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

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

Σ-*types*

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

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

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

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

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

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

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

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

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

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

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

Unit-types

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

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

Bibliography

[Agda] The Agda Development Team. *The Agda Programming Language*. 2020. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

[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.

[Bar91] Henk Barendregt. “Introduction to generalized type systems”. In: *Journal of Functional Programming* 1.2 (1991), pp. 125–154. DOI: 10.1017/S0956796800020025.

[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.

[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.

[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.

[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.

[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.

[Mim20] Samuel Mimram. *PROGRAM = PROOF*. Independently published, 2020. ISBN: 979-8615591839. URL: http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf.

[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.

[ML92] Per Martin-Löf. *Substitution calculus*. Notes from a lecture given in Göteborg. 1992.

[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.

[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`.

[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`.

[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`.