Crafting Provers in Racket

Adrian Manea
Coordinator: Assoc. Prof. Traian Şerbănuţă

April 20, 2020

Contents

INTRODUCTION AND MOTIVATION

The main motivation for starting the work on this project is my interest in the programming language Racket. This grew from me getting acquainted with Emacs and Lisps, from a user standpoint at first, but then my interest sparked further and I started reading more about lambda calculus and various other related subjects. Before long, I discovered John McCarthy's pioneering work in trying to make lambda calculus suitable for programming languages. But what really captivated me were the theoretical foundations that McCarthy's articles [17, 18, 19] set for this task, in a time when programming was either electrical engineering or pure mathematics. His work was said to have introduced a paradigm shift in computer science that is comparable to the non-Euclidean geometry revolution. The next step was discovering the influential "SICP" ([3]), which showed me how an apparently simple programming language such as Scheme can be used for wonderful constructions and various degrees of abstraction. This is further supported by the thorough specification and revisions that were published by the Scheme Community, the latest being [27].

Further reading in the world of Lisp dialects, I have discovered Racket, whose appeal was instantaneous to me, since it is so often described not as a general purpose programming language derived from Scheme[1], but rather as a toolbox for constructing languages to solve various problems. In fact, as I see it, it offers the necessary items for one to properly understand, craft and teach languages that exhibit particular behaviour.

This is precisely the aim of the current work. Having a background in mathematics, I quickly became interested on the one hand in proof assistants (or so-called "theorem provers") and type theory, on the other. The appeal of category theory mixed with (or, by some sources, morphed into) topos theory, type theory and the background of intuitionistic logic is very strong for me and I have been trying to approach the subject from many of its sides. Of equal interest for me is the teaching aspect. I believe in strong foundations, a thorough understanding of fundamentals before getting into more complex structures and I so often search for methods, tools and examples

[1] Racket used to be called PLT Scheme, where PLT is the name of the research group that has been working on this project since the very beginning, scattered throughout the world, as showcased at [1].

that I can use to showcase particular aspects of the subject I'm learning or teaching. Racket, for me, provides the perfect environment to fulfil most of such intentions. On the one hand, the language is big and flexible enough to be expressive for concepts of type theory, logic and proofs and on the other, it can be used in a piecemeal setup to serve as a great teaching toolbox for my purposes.

The plan of the work is as follows. We will start with some basic elements of type theory and intuitionistic logic. While the subject is hugely developed and used in many parts of (theoretical) computer science, we will try to make the work as self-contained as possible and as such, include in the preliminaries strictly the topics that will be developed further. For simplicity, but also for historical reasons, we will be focusing on the main work by one of the creators of the field, at least in terms of making its presentation appropriate for computer science applications, Per Martin-Löf, [14]. While Martin-Löf's lectures focus more on the theoretical side of things, the excellent [21] provides enough details to see how the concepts can be used in actual programming. However, in this work we will be presenting only the basics of typing judgments that are specific to Martin-Löf's type theory, as well as an introduction to untyped lambda calculus. Another theoretical feature of interest is that of dependent types, of which we sketch a brief introduction.

The second preliminary part of the dissertation will focus on Racket itself. We will try to provide a so-called "crash-course" to introduce some elements of syntax. Again, we will be focusing on items that will be used throughout the dissertation and for streamlining the content and since we appreciate that the syntax is quite self-explanatory, we decided to add it as an appendix.

A first, excellent example follows the theoretical part, along the lines of P. Ragde's article, [23]. There, the author showcases a toy proof assistant, called *Proust*, that they have written in Racket, the purpose being twofold. On the one hand, it shows the power that the language has for such tasks and on the other, as the author mentions, it helps them and their students to understand at least a part of the inner workings of well established proof assistants, such as Coq and Agda. That is, they will be implementing a small portion of the tools that are available in Coq and Agda in an intentionally verbose style so that their functions are transparent.

Finally, a more complex example is explored, that of *Pie*. Developed by Racket programmers, it is a domain-specific language as well as a proof assistant particularly featuring dependent types. We are convinced that presenting the syntax and some basic examples of Pie will not only help in understanding some fundamental characteristics of dependent types, but it will also show some concrete examples, focused on functional programming.

Apart from the references that are scattered thoughout the dissertation, our hope is that the material is self-contained and presented in such a way that it could be used as a teaching material at least for an introductory level, to spark the interest both in type theory and its implementation.

CHAPTER 1

MARTIN-LÖF TYPE THEORY

We start the theoretical part of this work focusing on Martin-Löf's take on intuitionistic logic and type theory. Before we do that, however, a quick note on the historical evolution of the subject. For brevity, we will use the abbreviation MLTT for Martin-Löf's Type Theory.

1.1 Brief Historical Overview

Type theory was first developed by Bertrand Russell and some of his collaborators and then expanded by Frank Ramsey and others. It was used first to introduce a kind of hierarchy of (mathematical) concepts that would "solve" Russell's paradox. At the same time, in the first part of the twentieth century, the Dutch mathematician L. E. J. Brouwer, then continuing with the American mathematician E. Bishop sowed the seeds of *constructive mathematics*. They put the emphasis on proofs that actually produce examples of the concepts existential quantifiers speak of. Thus, for example, any proof of a proposition of the form "$\exists x$ such that P" must contain a method of actually instantiating that x. This approach contrasted with the formalism developed by leading mathematicians such as D. Hilbert who proposed that mathematics should be performed simply following abstract rules and that showing a method to obtain an x in the previous example does not necessarily mean that that x should be obtained "practically", but only *in principle*.

At the same time, philosopher and logician A. Heyting summarized in his monograph [11] the approach that became known as *intuitionistic logic*. This emphasized constructive proofs as well and was used by Brouwer as a formal basis for his mathematical program.

In the second half of the twentieth century, the Swedish mathematician and philosopher Per Martin-Löf exposed his take on intuitionistic logic and type theory into what became known as *Martin-Löf type theory*. This approach draws influences from the philosophical work of F. Brentano, G. Frege and E. Husserl, but also uses the *Curry-Howard correspondence* between propositions and programs, terms and proofs (excellently detailed in [26]).

Martin-Löf's approach became extremely influential and with the help of works such as [21], it was quickly implemented as a foundation for extremely powerful proof assistants such as NuPRL, Coq, Agda, Idris and others.

It is also worth mentioning that type theory and intuitionistic logic have continued to develop independently from applications in computer science. As such, types and toposes are seen as good candidates to provide "proper" foundations of mathematics instead of sets. Some very important contributions have come from the HoTT group ([10]).

We will not go into more historical or philosophical details of this subject and instead further focus on the mathematical aspects that were then implemented in proof assistants.

It is worth mentioning, however, that type theory (along with topos theory) has evolved into a domain which claims to serve as a better foundation for mathematics than set theory. As such, there are significant differences between sets and types, which are excellently summarized at [10, §1.1].

1.2 Preamble: Set Theory and Type Theory

Before we get to more technical details, we make a short stop to discuss some specific differences between set theory and type theory. We should point out that many of the type-theoretical and logical aspects of homotopy type theory (hereafter referred to as HoTT) are taken from Per Martin-Löf's work, some of which we touch on below. Nevertheless, given the complexity and the size of the HoTT project, multiple changes have appeared, some more radical, some only to modernize the presentation.

The main starting point is that type theory aims to provide foundational results for mathematics, overcoming the difficulties that set theory was shown to contain[1]. For this reason, many discussions have been sparked as to what differentiates type theory from all the previous attempts of formalizing the foundations of mathematics in set theory. Without going into technical, complex discussions, we mention some basic points that are relevant to the current presentation. We rely for this on the overview presented in [10, §1.1].

First, it should be pointed out that set theory, as used in the foundational research, is comprised of two parts:

- the deductive layer of first order logic, that is used to reason, i.e. to write proofs;
- the axioms that are formulated in this system and which regulate the objects that live in the theory. In this case, the Zermelo-Fraenkel with the Choice Axiom (usually written as ZFC) is the most used axiomatic system.

[1] On a similar note, *topos theory* is another approach that has a similar goal, coming from topology and category theory. While the subject is way beyond the scope of this dissertation, we mention it and refer the interested reader to the excellent book [13] or the more introductive [9]. A historical overview of the topic is presented, for example, in [20].

It follows that set theory is not actually about sets *per se*, but rather about the interactions between the sets, as objects instantiated by the second (axiomatic) layer by the rules of the first (deductive) layer, which help us make propositions.

On the other hand, type theory is built upon itself, in a way. What we mean is that type theory is its own deductive system. Whereas set theory is based on *sets* and *propositions* (about sets), type theory is only about *types*. As such, the primitive notion of type can be subject to various interpretations which become (parts of) mathematical theories: some types can be seen as propositions, some as sets, some as topological spaces, categories and so forth. It follows that the mathematical (*formal*) act of *proving a theorem* inside set theory becomes the *constructive* (i.e. intuitionistic) act of *constructing an object*, namely an inhabitant of a type.

Another key difference is related to the structure of deductions themselves. In general, we can make a deductive system from (inference) *rules* and *judgments*, which are derived by the rules. In the case of first order logic, which regulates set theory, the basic judgment is that a proposition has a proof. From it, one can construct derived proofs, such as, for example, the proof of $A \wedge B$, which is made of a proof of A in conjunction with a proof of B, by a rule of proof construction.

On the other hand, the basic judgment of type theory is of the form $a : A$, read "*element a has type A*". But this judgment is analogous to the provability of proposition A, were it to be interpreted as a type. That is, exposing an inhabitant a of the type A *is not* similar to showing an element of a set $a \in A$. Membership is taken for granted in type theory when we make a typing judgment. That is, the judgment $a : A$ cannot be proved or disproved so it cannot be used in composite judgments such as *if* $a : A$ *then* $b : B$. Intuitively, this is because whenever we want to model the use of a certain element, we think of its "natural type", i.e. one which we are sure it inhabits, then start from there. We never, even in common mathematical speech, start with something like *Let* $a = \dfrac{\sqrt[3]{1+\sqrt[5]{5}}}{\ln 7} : \mathbb{N}$ if we are not sure that it is true first. Usually, if we need to use a in some proof, we start with it as an inhabitant of a type which we are sure it belongs to, e.g. *let* $a : \mathbb{R}$[2].

Lastly, a very important difference between sets and types is the treatment of *equality*. In common set theory, equality is a proposition, meaning that it can be proved or disproved for various sets or elements. In type theory, however, *equality is a type*. That is, taking, say $a, b : A$, there exists the type $a =_A b$, which is inhabited if and only if a and b represent the same element. Before going further, a quick detour: What would an inhabitant of an equality type look like? What "instantiates" the fact that is commonly written as $a = b$? This is not an easy question and we only mention that this is one of the starting points of the *homotopy* part of HoTT. That is, using Martin-Löf's brilliant idea of making equality a type, homotopies (formally known as *homotopy maps*) entered the scene. They provide a kind of "relaxed" form of equality by means of, e.g. in topology, *continuous deformations*. Since in topology, continuity is one of the most important properties, if a space can be continuously transformed into another one, we simply say that "they

[2]Note that we have not provided examples of types yet and if this example induces the confusion that all sets are types or vice versa (or at least, numeric sets), feel free to skip this discussion, as we will not insist on it anyway.

are the same"[3] What this example shows is that there are at least some interpretations of type theory where (the local notion of) equality can have witnesses, which are, in this case, *homotopy maps*.

Back to the original discussion, when the type $a =_A b$ is inhabited, we say that the elements a and b are *propositionally equal*. That is, the fact that an equality type is inhabited provides an example of propositional equality. But there is another kind of equality, called *judgmental* or *definitional* equality, which is more akin to the common mathematical sense. For example, 3^2 and 9 are equal by definition (of multiplication, in this case), so they exhibit a definitional equality. It is *judgmental* in the sense that it can be proved, e.g. by using the definition of multiplication in the example we provided. We finish by mentioning that this kind of equality can be used for whole function equality, for example, by means of the distinction between intensional and extensional concepts which we mention in Remark 1.1. For example, the function $x \mapsto (x+2)$ is *judgmentally equal* to the function $y \mapsto (y+3-1)$. Usually, definitional equality is denoted by $\equiv$, so $3^2 \equiv 9$ etc.

1.3 Fundamentals of MLTT

MLTT draws inspiration from *Gentzen natural deduction system* from the 1930s, explained in more modern terms in [8]. As such, judgments will be written in the so-called *proof tree form*, such as:

$$\frac{A}{A \vee B}$$

where above the line we have the hypotheses and below the inferences. In particular, the above rule takes for granted that we have some formulas A and B and it infers that $A \vee B$ is true given the hypothesis that A is true.

Given some A, which can be a set or a proposition, we write $a \in A$ to mean, using the Curry-Howard correspondence, that either:

- a is an element of the set A;
- a is a proof of the proposition A.

Also, the judgment $a = b \in A$ means more than meets the eye:

- A is a proposition or a set;
- a and b are proofs or elements respectively;
- a and b are identical elements of the set A or represent identical proofs of the proposition A.

In fact, more than these readings can be given for the simple judgment $a \in A$, as shown in the table of figure 1.1.

[3] This is the origin of the famously funny "facts" which state that a bagel is the same as a cup or a cow is the same as a sphere.

A set	$a \in A$	implies
A is a set	a is an element of the set A	A is nonempty
A is a proposition	a is a (constructive) proof of A	A is true
A is an intention (expectation)	a is a method of fulfilling A	A is fulfillable (realizable)
A is a problem (task)	a is a method of solving A	A is solvable

Figure 1.1: Readings of the judgment $a \in A$, cf. [14, p. 4]

In particular, the reading which refers to problem-solving is commonly attributed to A. Kolmogorov ([12]) and called the *Brouwer-Heyting-Kolmogorov interpretation*.

What is characteristic of MLTT is that in order to ascertain that we have a set (proposition), we must prescribe an *introduction* rule, by means of showing how a *canonical* element of the set (proof of the proposition, respectively) is constructed an *equality* rule which shows how we know that two canonical elements are equal.

For example, for the set of positive integers (using the Peano approach and denoting by a' the successor of the element a), we can give the rules:

- for the canonical elements: $0 \in \mathbb{N}$ and $\dfrac{a \in \mathbb{N}}{a' \in \mathbb{N}}$;
- equality of canonical elements: $0 = 0 \in \mathbb{N}$ and $\dfrac{a = b \in \mathbb{N}}{a' = b' \in \mathbb{N}}$.

Another example, for the product of two sets (propositions) $A \times B$, we have:

- canonical elements:
$$\frac{a \in A \qquad b \in B}{(a, b) \in A \times B};$$
- equality of canonical elements:
$$\frac{a = c \in A \qquad b = d \in B}{(a, b) = (c, d) \in A \times B}$$

Now, equality of the sets (propositions) as a whole is prescribed by showing how equal and canonical elements are formed for the sets (propositions). For example, for sets (propositions), the equality is simply:
$$\frac{a = b \in A}{a = b \in B}.$$

For non-canonical elements, to explain what it means for them to be elements of a set (proposition) A, we must specify a method (proof, program) which, when executed (performed), it yields a *canonical* element of the set as a result.

Then, finally, two arbitrary (not necessarily canonical) elements of a set A are equal if, when executed as above, yield equal *canonical* elements of the set.

It is now worth focusing our attention on the particular cases of propositions. Given the setup above, we can write the table in figure 1.2.

a proof of	consists of
$\bot$	—
$A \wedge B$	a proof of A and a proof of B
$A \vee B$	a proof of A or a proof of B
$A \supset B$	a method taking any proof of A into a proof of B
$(\forall x)B(x)$	a method taking any individual a into a proof of $B(a)$
$(\exists x)B(x)$	an individual a and a proof of $B(a)$

Figure 1.2: Proofs of propositions, cf. [14, p. 7]

We can be more precise than that, noting further:

- the proposition $\bot$ has no possible proof;
- (a, b) is a proof of $A \wedge B$, provided that a is a proof of A and b is a proof of B;
- $i(a)$ or $j(b)$ are proofs of $A \vee B$, provided that a is a proof of A and b is a proof of B, where i and j are the canonical inclusions (which we detail later);
- $\lambda x.b(x)$ is a proof of $A \supset B$, provided that $b(a)$ is a proof of B under the hypothesis that a is a proof of A;
- $\lambda x.b(x)$ is a proof of $(\forall x)B(x)$, provided that $b(a)$ is a proof of $B(a)$, where a is some individual;
- (a, b) is a proof of $(\exists x)B(x)$, given that a is an individual and b is a proof of $B(a)$.

Also, the presentation can be extended further to *types*. From a historical, philosophical and mathematical point of view, types themselves are taken as primary notions, hence not properly defined. Intuitively though, one can think of types as "hierarchies", "levels" of certain kinds of elements or rather use the computer science intuition of *data types*.

For this case, the judgment "a is an element of type A" is commonly denoted by $a : A$ and if this is the case, we say that the type A is *inhabited*.

It can be formally shown that the type-theoretical approach is isomorphic to the propositional approach and the set-theoretical approach, making what is commonly known as the *formulas-as-types* or *propositions-as-sets* interpretations (for intuitionistic logic).

1.4 Untyped Lambda Calculus

Since in the Proust application that we present in §2 we will build a small proof assistant to verify proofs based on untyped lambda calculus, we will spend some space here to rigorously introduce some basic notions of this formalism. The simply typed version is not much more complicated and we will try to cover it briefly in the next section, in the context of dependent types.

This short presentation follows [22, §5]. Much details, along with its connection to propositional logic, can be found in [26].

Therefore, the grammar of the untyped lambda calculus can be described in BNF as below:

$$t ::= x \mid \lambda x.t \mid tt,$$

meaning, respectively, variables, lambda abstractions and applications.

Two keywords are essential at this point:

- *metavariables* are the variables that are abstracted. So for example, in the expression $\lambda x.x^2$, x is a metavariable, since it can be renamed (consistently) without losing any meaning of the term. That is, $\lambda x.x^2$ is the same term as $\lambda y.y^2$, for example;
- the *scope* of variables is the region of the expression where it is *bound*. In the lambda term above, the variable x is bound in the whole body of the expression, λx being the actual binder. But in the term $\lambda x.\lambda y.x \cdot y$, x is bound in both the interior and the exterior lambda, while y is bound only in the interior expression. Finally, in the term $\lambda x.xy$, y is free.

  A term that has no free variables is called a *combinator*. The simplest example is the *identity function*, $\lambda x.x$.

We should also note that lambda application *associates to the left*. Hence, for example, in the expression:

$$(\lambda x.\lambda y.xy)ab,$$

we first bind x to a.

Although seemingly simple, untyped lambda calculus can be used to express some essential programming features, such as Booleans, numerals and branching expressions. They are usually called with A. Church's name as a prefix, i.e. *Church Booleans, Church numerals* and (less common) *Church branching*.

For this purpose, we introduce:

$$\texttt{true} = \lambda t.\lambda f.t$$
$$\texttt{false} = \lambda t.\lambda f.f.$$

To test this, we introduce a combinator `test`, which is basically a branching expression. That is, let:

$$\texttt{test} = \lambda l.\lambda m.\lambda n.lmn.$$

We will see how this expression reduces to m if l is `true` and it reduces to n if l is `false`. That is, we can understand the expression as:

$$\texttt{test} = \texttt{if l then m else n}.$$

Here is an example computation:

$$\begin{aligned}
\texttt{test true v w} &= (\lambda l.\lambda m.\lambda n.lmn)\texttt{true}vw \\
&\longrightarrow (\lambda m.\lambda n.\texttt{true}mn)vw \\
&\longrightarrow (\lambda n.\texttt{true}vn)w \\
&\longrightarrow \texttt{true}vw \\
&= (\lambda t.\lambda f.t)vw \\
&\longrightarrow (\lambda f.v)w \\
&= v.
\end{aligned}$$

More expressions can be defined, such as:

$$\begin{aligned}
\texttt{and} &= \lambda b.\lambda c.bc\texttt{false} \\
\texttt{pair} &= \lambda f.\lambda s.\lambda b.bfs \\
\texttt{first} &= \lambda p.p\texttt{true} \\
\texttt{second} &= \lambda p.p\texttt{false}.
\end{aligned}$$

They are detailed at [22, pp. 59-60].

Finally, we reach *Church numerals*. They are lambda expressions which "act like numbers". That is, when the numeral c_n is fed as an argument to a function, it makes the function apply n times. Here are the definitions of the first few:

$$\begin{aligned}
c_0 &= \lambda s.\lambda z.z \\
c_1 &= \lambda s.\lambda z.sz \\
c_2 &= \lambda s.\lambda z.s(sz) \\
c_3 &= \lambda s.\lambda z.s(s(sz))
\end{aligned}$$

Notice the suggestive naming of variables, z reminding of zero and s reminding of the successor function. Another remark is that c_0 is actually `false`, with its variables renamed.

But we can define the successor function for Church numerals as well:

$$\texttt{succ} = \lambda n.\lambda s.\lambda z.s(nsz).$$

When applied to a Church numeral c_p, it will produce the expression of the succeeding Church numeral c_{p+1}.

We won't get into any more details here, for multiple reasons. On the one hand, the theory is reach enough to be hard to cover it extensively in this dissertation, therefore pointers to literature suffice for our purposes. On the other hand, we will not be concerned with more technical results or complications in what follows, so what we presented so far should serve as a reasonable prerequisite. We are also convinced that should novelties appear, they will be easy to grasp on the go. As we mentioned, we will be focusing on untyped lambda calculus only in presenting the Proust "nano prover" in §2.

1.5 Dependent Types — Judgments and Inference Rules

In many cases, it is very useful to have *dependent types*, namely types which, in a way, are *parametrized* by other types. A particular illustrating example is when we want to define, say, a function that selects the first element of a list, but we don't want to define it separately for lists of integers, then for lists of strings, then for lists of floats etc. We just want one function that does the job regardless of the types of the arguments.

A similar situation may be familiar from elementary mathematics. For example, if we have a *sequence of functions*, such as:

$$f_n : \mathbb{R} \longrightarrow \mathbb{R}, n \in \mathbb{N}$$

and each of the terms of the sequence is a different function, e.g.:

$$f_1(x) = \cos x, \quad f_2(x) = x^2, \quad f_3(x) = \exp(x) \dots,$$

we can say that we have *parametrized* the function f by the positive integers n which serve as the indices of the sequence.

Now imagine that the indices can be of different types, e.g. we can have $f_{a[]}$ and f_5 and $f_{'a'}$ in the same sequence and depending on the case, we know how to define the function. For example:

- for any $a \in \mathbb{Z}$, define $f_a = a^2 - 3a + 1$;
- for any vector `a[]`, define $f_{\texttt{a[]}}$ to be the sum of the elements of `a[]`;
- for any character `c`, define $f_{\texttt{c}}$ to be the ASCII code of `c`.

What we have "defined" in the toy example above is actually a *dependent function* which has one extra argument that is not obvious. So to be precise, we are defininining something like $f(x, T)$ (also written as $f(T)(x)$ or $f_T(x)$), where T is the type of x and depending on this T we actually know how to compute the value of the function in x accordingly. Once we have a fixed type T, the function is computed with a uniform formula for all arguments. So in the example above, the three rules that separate the cases for the types of the argument are each uniform: for *any integer* we compute that polynomial expression, for *any vector* we take the sum of its elements and for *any character*, we take its ASCII code.

While this may seem complicated and the example above may not be the most illuminating, we emphasize once again that dependent types are extremely useful for the case when we want to define functions that work in a similar way for various types. For example, we can modify the definitions above so that they do similar things regardless of the type of the argument:

- for any integer $a \in \mathbb{Z}$, define $f_a = a^2 - 3a + 1$;
- for any vector `a[]`, define $f_{\texttt{a[]}}$ to first compute the sum of the elements in `a[]`, store it in `s` and then compute $s^2 - 3s + 1$;
- for any character `c`, define $f_{\texttt{c}}$ to first take the ASCII code of `c`, store it in c, then compute $c^2 - 3c + 1$.

In such a situation, the function acts *as if* it is the polynomial $X^2 - 3X + 1$, computed *regardless* of the types of its arguments (integers, vectors, characters). That is, we have implemented a sort of "general polynomial" whose definition is *dependent* on the type of the argument, but the overall look is "the same".

These are the basic ideas which can also serve as motivations for what follows. We now go into a more rigorous presentation, following [25].

As it was seen in the theory developed by Martin-Löf, it is fundamental to specify what kinds of types, terms and judgments are primitive to one theory. In the case of the dependent type theory, we start with four primitive judgments:

(1) A is a well formed *type* in a context Γ, written as:

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

(2) A and B are *judgmentally equal types* in a context Γ, written as:

$$\Gamma \vdash A \equiv B \text{ type};$$

(3) a is a well-formed *term* of type A in a context Γ, in symbols:

$$\Gamma \vdash a : A;$$

(4) a and b are *judgmentally equal terms* of type A in a context Γ, in symbols:

$$\Gamma a \equiv b : A.$$

In all the above, a *context* is just an expression which puts together all we assume to know so far. In symbols, it can be written as:

$$\Gamma = x_1 : A_1, x_2 : A_2(x_1), \dots, x_n : A_n(x_1, \dots, x_{n-1}),$$

and interpreted as being made of the statement $x_1 : A_1$ (where x_1 is a well-formed term and A_1 is a well-formed type), then $x_2 : A_2$, *but* knowing already information about x_1 and so forth, until the last statement, which presupposes the previous $n - 1$ statements.

Most of the times, we will be omitting the presuppositions, in the sense that we will just write $x_1 : A_1, \dots, x_n : A_n$. However, the presuppositions are essential, since they actually mean, for any $1 \leq k \leq n$ that:

$$x_1 : A_1, x_2 : A_2, \dots, x_{k-1} \vdash A_k \text{ type},$$

namely that they all make up the necessary hypothesis which allows us to conclude that the next item is a well-formed type (under this hypothesis).

Another item of terminology is that we say that the context Γ above *declares the variables* $x_1, \dots, x_n$. An *empty context* declares no variable and a type that is well-formed in an empty context will be called a *closed type*, as well as a well-formed of such a type will be called a *closed term*.

Getting towards dependent types, we may enrich a context with one more declaration then obtain a judgment there, such as:

$$\Gamma, x : A \vdash B \text{ type}. \tag{1.1}$$

This means that if we enlarge the context Γ with the declaration $x : A$ we can then judge that B is a well-formed type, which will be called a *type dependent on A*. An alternate name for B is actually a *family of types* over A (in context Γ), since by changing x, we can change A, which in turn could change B, but such that the judgment in (1.1) remains valid.

We should also point out that we have been emphasizing *judgments* and *judgmental* equalities. They contrast *definitions* and *definitional* equalities respectively in the sense that judgments are *derived*, whereas definitions are postulated.

Hence, in order to derive judgments, we need *inference rules*. They actually say that *judgmental equality is an equivalence relation* between judgments:

- reflexivity for types: $\dfrac{\Gamma \vdash A \text{ type}}{\Gamma \vdash A \equiv A \text{ type}}$;
- reflexivity for terms: $\dfrac{\Gamma \vdash a : A}{\Gamma a \equiv a : A}$;
- symmetry for types: $\dfrac{\Gamma \vdash A \equiv A' \text{ type}}{\Gamma \vdash A' \equiv A \text{ type}}$;
- symmetry for terms: $\dfrac{\Gamma a \equiv a' : A}{\Gamma \vdash a' \equiv a : A}$;
- transitivity for types: $\dfrac{\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma \vdash A' \equiv A'' \text{ type}}{\Gamma \vdash A \equiv A'' \text{ type}}$;

- transitivity for terms: $\dfrac{\Gamma \vdash a \equiv a' \,:\, A \qquad \Gamma \vdash a' \equiv a'' \,:\, A}{\Gamma \vdash a \equiv a'' \,:\, A}$.

Aside from these basic rules, we also have *structural rules*, which can be used in more complex derivations. They specify how *weakening*, *substitutions* and *use of variables* are performed.

(1) *Weakening a context:* As in the case of any deductive thinking, to weaken a hypothesis means to add more information to it (as opposed to strengthening it by removing information). Hence, the rule for weakening a context (denoted by W_A) is the following:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, \Delta \vdash J}{\Gamma, x \,:\, A, \Delta \vdash J} W_A.$$

the explanation of the rule is as follows. We start with a context Γ where we make a typing judgment about A. Then we also assume we have a larger context, comprised of Γ and Δ in which we have a judgment J (which can be any of the four kinds introduced earlier). We conclude that in a weakened context where we include both Γ and Δ, as well as a typing judgment which makes use of A, namely the declaration of the *fresh variable* $x \,:\, A$, we can still judge J.

(2) *Variable rule:* This rule basically introduces an identity function for any type A, denoted by δ_A:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma, x \,:\, A \vdash x \,:\, A} \delta_A. \tag{1.2}$$

What we're seeing is that starting from a well-formed type, the judgment of a well-formed term of that type is automatic if we enlarge the context to contain that term as well. The identity function basically maps identically the $x \,:\, A$ from the hypothesis (context) to the conclusion (judgment).

(3) *Substitution rule:* We know already that we can rename variables consistently without changing anything in the judgments. This rule shows that we can perform substitutions consistently and this well preserve well-formedness of types, terms and judgmental equality (for types and terms):

$$\frac{\Gamma \vdash a \,:\, A \qquad \Gamma, x \,:\, A, \Delta \vdash J}{\Gamma, \Delta[a/x] \vdash J[a/x]} S_a.$$

This rule of substituting a for x basically says that we can safely substitute x with a consistently in a context and what will happen is that the substitution will propagate to the initial judgment.

The variations of this rule are for terms and types, which we write below and assume they are self-explanatory, following the explanations above:

$$\frac{\Gamma \vdash a \equiv a' \,:\, A \qquad \Gamma, x \,:\, A, \Delta \vdash B \text{ type}}{\Gamma, \Delta[a/x] \vdash B[a/x] \equiv B[a'/x] \text{ type}};$$

$$\frac{\Gamma \vdash a \equiv a' : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \equiv b[a'/x] : B[a/x]}.$$

Taking from geometry, when B is a family of types over A and we have some $a : A$, we call $B[a/x]$ *the fiber* of B at a and sometimes denote it shorter with $B(a)$.

1.6 Dependent Π Types

How do we actually "parametrize" a type in general by another type? And assume we did that. What are the well-formed terms that inhabit such types? We are about to find the answers to these questions: the types are called *(dependent) function types* or Π-types and their terms will be lambda abstractions.

Again taking from Martin-Löf, in order to specify a type, it is enough to give its *formation rule* (called *introduction* by Martin-Löf) and a rule which shows how to identify identical types of this sort. They are, respectively, the Π-formation rule and the Π-equality rule, presented below:

- Π formation: $\dfrac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \Pi_{(x:A)} B(x) \text{ type}} \Pi - \text{intro};$
- Π-equality: $\dfrac{\Gamma : A \equiv A' \text{ type} \qquad \Gamma, x : A \vdash B(x) \equiv B'(x) \text{ type}}{\Gamma \vdash \Pi_{(x:A)} B(x) \equiv \Gamma_{(x:A')} B'(x) \text{ type}} \Pi - \text{eq}.$

In words, if we start with a context Γ and weaken it with a declaration of a fresh variable $x : A$ (assumed arbitrary) and if in this (weakened) context we have a well-formed type which is denoted by $B(x)$ precisely to emphasize that it holds for any x, the conclusion is that we can form a type which engulfs this arbitrary x and the respective family $B(x)$. This is the so-called *product type* or Π-type and is denoted by $\Pi_{(x:A)} B(x)$. Again, we emphasize the fact that this holds for all x which is fixed for a particular type, but can be varied freely and the judgment still holds.

The situation is not dissimilar to the one we explained intuitively at the beginning of the previous section: we have some judgment (in this case, the well-formedness of the type $B(x)$) which is parametrized by some x in the sense that it can be different when changing x, but what's not different is that the judgment of well-formedness still holds true. So basically we have a sort of a *family of judgments*, that are accounted by the possible change of x.

This still holds true if we uniformly rename the parameter x. That is, assuming that $x' : A$ is a fresh variable that does not occur in the hypothesis $\Gamma, x : A$ (i.e. $x \notin \Gamma$ and $x' \neq x$), we have the Π-renaming rule:

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \Pi_{(x:A)} B(x) \equiv \Pi_{(x':A)} B(x') \text{ type}} \Pi - x'/x.$$

The well-formed terms that inhabit such Π-types are special cases of λ-abstractions. Their special quality consists in that they can produce outputs of different types, *depending* (pun intended!) on the type of the input, but they do this in a uniform way, in the sense that it can be

captured in an inference rule:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.b(x) : \Pi_{(x:A)}B(x)}\lambda_A.$$

Notice how we specified everywhere that both the terms b and the types B *depend* on the choice of $x : A$.

The term $\lambda x.b(x)$ outputs a well-formed term of the product type. Intuitively, it is a general rule that can be applied "in the same way" regardless of the types of its input and which can produce different types of output.

For example, we are taught in elementary geometry that two plane vectors, understood as drawn arrows, are perpendicular if we superimpose a geometric tool for measuring angles and we see the indication of 90°. But then in high school, we learn that vectors are also pairs of reals and perpendicularity can be translated in terms of the dot product being null. Then in college, abstraction increases and we see that the same dot product can be adapted (almost identically) to functions, matrices, polynomials and other algebraic structures in the context of Euclidean vector spaces. Now we realize we know a *dependent function:* the dot product. It is computed "essentially the same" regardless of the type of its inputs: pairs or triples of reals, matrices, continuous functions, polynomials etc. The output is still a real number, so the output type does not change once the input type changes, but this is irrelevant. The basic idea is to have a uniform (rule-like) method of associating something functionally to an input, regardless of its type.

What follows now is to give a rule which enables us to infer that two lambda abstractions are identical:

$$\frac{\Gamma, x : A \vdash b(x) \equiv b'(x) : B(x)}{\Gamma \vdash \lambda x.b(x) \equiv \lambda x.b'(x) : \Pi_{(x:A)}B(x)}\lambda_A - \text{eq}$$

What this rule says is that if we start with identical terms (more precisely, *pairs* of identical terms, one pair for each parameter x), then the lambda abstractions that associates to the parameter x any of these terms are themselves identical.

Remark 1.1: A technical word of caution is appropriate here. Both in mathematics and in philosophy, we have the complementary terms of *intensional* and *extensional* equality or definitions. Their distinction is relevant here.

We call the *extension* of a concept the collection of examples or items that are characterized by that concept. For example, if the concept is "cuteness", all the cute objects belong to the extension of that concept. Mathematically, the extension of a functional concept (i.e. of a function) $f : A \longrightarrow B$ is the image of the function, namely $f(A)$.

The *intension* of a concept is the actual definition of the concept, seen in the most abstract way possible. Mathematically, the intension of a functional concept (a function) $f : A \longrightarrow B$ is the actual definition that makes up the function. So, say, if $f(x) = x^2$, the intension of f is "squaring the argument".

Now, for functions, which are one of the core concepts of this work, we have two kinds of equality:

- *extensional equality*, which says that two functions $f, g : A \longrightarrow B$ are equal iff $f(x) = g(x), \forall x \in A$, namely if $f(A) = g(A)$, i.e. their extensions coincide (as sets). This is usually called in mathematics *pointwise equality*.

- *intensional equality*, which makes two functions as above equal iff their actual definitions coincide (perhaps after algebraic manipulation). In mathematics, this is usually called *identity*, a very strong and rare relation between nontrivial objects. So for example, the functions $x \mapsto x^2$ and $y \mapsto (y + 1 - 1)^2$ are intensionally equal (they are also extensionally equal).

In general, the two notions are not the same, although it is not easy to provide counterexamples.

The point of this remark is that the rule λ_A – eq is an item of *intensional equality*, since we're assuming that the output expression of the two lambda terms are the same (as it was the case of x^2 and $(y + 1 - 1)^2$ in our example).

This distinction used to be central in the early days of type theory, especially when it was infused with philosophy (more precisely, logicism). But in (theoretical computer science) practice, intensional concepts are usually preferred, as they provide a "tighter" form of identification.

Back to the λ_A–eq rule, we also have the variation which uses fresh variables. Hence, if $x' : A$ is fresh, i.e. $x' \neq x$ and $x' \notin \Gamma$, then we have:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.b(x) \equiv \lambda x' b(x') : \Pi_{(x:A)}B(x)}\lambda_A - x'/x.$$

Whenever it is possible, we will still use the more common notation for functions, hence a *dependent function*, which is a well-formed term of a Π-type will be denoted either explicitly by the lambda notation, such as $\lambda x.b(x) : \Pi_{(x:A)}B(x)$, or implicitly, by $f : \Pi_{(x:A)}B(x)$, being understood that the argument of f is x.

The actual use of functions is contained in the *evaluation rule*:

$$\frac{\Gamma \vdash f : \Pi_{(x:A)}B(x)}{\Gamma, x : A \vdash f(x) : B(x)}\mathrm{ev}_A.$$

Moreover, basic conversions can be performed inside functional terms, both of implying that evaluation and lambda abstraction are mutually inverse operations:

- β-reduction: $\dfrac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma, x : A \vdash (\lambda y.b(y))(x) \equiv b(x) : B(x)}\beta$. This shows how λ expressions work, in general, so nothing special here.

- η-conversion: $\dfrac{\Gamma \vdash f : \Pi_{(x:A)}B(x)}{\Gamma \vdash \lambda x.f(x) \equiv f : \Pi_{(x:A)}B(x)}\eta$, which says nothing else that another name for a lambda abstraction of the form $\lambda x.f(x)$ is f.

Note that as in the case of regular (untyped) lambda calculus, functions of multiple arguments can be written as iterating lambda abstraction (which in turn produces terms of iterated Π-types). As such, a function of two arguments $(x, y) \in A \times B$ can be written as $\lambda x.\lambda y.f(x, y)$ or simply $\lambda xy.f(x, y)$, which is a term of type $\Pi_{(x:A)}\Pi_{(y:B)}B(x, y)$.

1.7 Function Types

We can obtain such types as a particular case of Π types. As such, assume A and B are both types in context Γ. First we weaken B by A (i.e. add A to the context), then form the corresponding Π-type:

$$\frac{\dfrac{\Gamma \vdash A\ \mathrm{type} \qquad \Gamma \vdash B\ \mathrm{type}}{\Gamma, x : A \vdash B\ \mathrm{type}}}{\Gamma \vdash \Pi_{(x:A)}B\ \mathrm{type}}.$$

What we achieve by this is that we somehow *fix* the type A in the intermediate step where we weakened the context Γ, since now A (by means of all its arbitrary inhabitants $x : A$) is added to the hypothesis. This means that such particular Π types contain ordinary functions $A \to B$, which is also the notation that we use for this *function type*.

It follows that all the inference rules for this type can be obtained as a particular case of Π types, so we just list them accordingly:

- $\to$-introduction (formation): $\dfrac{\Gamma \vdash A\ \mathrm{type} \qquad \Gamma \vdash B\ \mathrm{type}}{\Gamma \vdash A \to B\ \mathrm{type}}\ \to\text{-intro}$;
- lambda abstraction: $\dfrac{\Gamma \vdash B\ \mathrm{type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma \vdash \lambda x.b(x) : A \to B}\lambda$;
- evaluation: $\dfrac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B}\mathrm{ev}$;
- β-reduction: $\dfrac{\Gamma \vdash B\ \mathrm{type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma, x : A \vdash (\lambda y.b(y))(x) \equiv b(x) : B}\beta$;
- η-conversion: $\dfrac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x.f(x) \equiv f : A \to B}\eta$.

Note that we explicitly omitted mentioning A in any of the rules (i.e. we wrote λ and not λ_A for the rule of λ-abstraction, similarly for evaluation), since as explained at the beginning of this section, function types are obtained precisely by *fixing* A, so it is understood.

Now we can obtain the proper identity function of any type A by using the variable rule (1.2):

$$\frac{\dfrac{\Gamma \vdash A\ \mathrm{type}}{\Gamma, x : A \vdash x : A}}{\Gamma \vdash \mathrm{id}_A :\equiv \lambda x.x : A \to A}.$$

We won't be getting into more complex constructions with dependent types, but we only mention that function composition is now easy to define:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type} \qquad \Gamma \vdash C \text{ type}}{\Gamma \vdash \text{comp} : (B \to C) \to ((A \to B) \to (A \to C))}.$$

In particular, the common mathematical notation $g \circ f$ will mean $\text{ev}(\text{ev}(\text{comp}, g), g)$ in this context.

1.8 Example: The Natural Numbers

As a closing theoretical example of this chapter, we show how the natural numbers can be formed as a type, using the Peano triple and the induction principle.

Therefore, we introduce $\mathbb{N}$, the *type of natural numbers*, which is a closed type, equipped with two special *closed terms*, one for zero and another one, for the successor function:

$$0 : \mathbb{N}, \quad S : \mathbb{N} \to \mathbb{N}.$$

Next, the *induction principle*:

$$\frac{\Gamma, n : \mathbb{N} \vdash P \text{ type} \qquad \Gamma \vdash p_0 : P(0) \qquad \Gamma \vdash p_S : \Pi_{(n:\mathbb{N})} P(n) \to P(S(n))}{\Gamma \vdash \text{ind}_{\mathbb{N}}(p_0, p_S) : \Pi_{(n:\mathbb{N})} P(n)} \mathbb{N}\text{ – ind.}$$

In words, this says that if we start with a type and a fixed natural n in a certain context (understood as a set of hypotheses) such that in that context, the (dependent) type $P(0)$ is inhabited by a term p_0 and the (dependent) function type $P(n) \to P(S(n))$ is also inhabited by a term p_S, then the type $P(n)$ will be inhabited. As this latter type is also dependent, it could be read as "$P(n)$ is inhabited for all $n : \mathbb{N}$".

With these at hand, we can define *addition*:

$$\text{add} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

by induction. It means that we need to construct a function $\text{add}_0 : \mathbb{N} \to \mathbb{N}$ (for the base case) and another function for the inductive step:

$$\text{add}_S(f) : \mathbb{N} \to \mathbb{N},$$

under the hypotheses $n : \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$.

This is easy: take add_0 to be $\text{id}_{\mathbb{N}}$, the identity function, since it adds nothing to its argument (actually, it does add, *zero*) and then define the inductive step to be:

$$\text{add}_S(f) = \lambda m.S(f(m)),$$

which basically means that we are adding one to f pointwise (i.e. $\text{add}_S(f)(m) = (f(m) \mapsto f(m)+1)$).

We can derive this formally, but we skip the details and refer the reader to [25, pp. 12-13].

So what we get is a general addition function, add which works as expected:

$$\text{add}(n, m) = n + m$$

and using the basic terms of type $\mathbb{N}$, we can derive:

$$0 + m \equiv m, \quad S(n) + m \equiv S(n + m).$$

We stop here with the theoretical presentation and examples, pointing the reader to the article [16] for another great reference that contains both excellent theoretical insight, as well as illuminating practical examples in functional programming.

Furthermore, the implementation of most elements of Martin-Löf's type theory is presented in a very readable manner in [21].

CHAPTER 2

PROUST

This chapter will provide an example of a great use of Racket to craft a "nano proof assistant", called Proust. The presentation closely follows [23] and it will contain further clarifications as needed. For a very basic Racket introduction, we have added in the Appendix some special syntactic features and we also point the reader to the excellent official documentation ([2]).

As the author mentions, the article is intended as a DIY approach for a simple proof assistant (in fact, the name derives from a weird contraction of the expression "proof assistant"). The general goal is much more interesting than that, in that it will also delve into the inner works of the underlying mechanisms of proof asistants, for the purpose of implementing them in a simpler manner.

Note that since Racket is rather a toolbox for crafting one's own toy languages, any source code must start with a `#lang` pragma, which mentions what part of Racket one wants to use. Common pragmas (formally, *modules*) include:

- `htdp` – the special module with syntax used in the [6] book;
- `racket` – the full-fledged module with all syntax and features;
- `racket/base` – the simplest submodule of `racket`;
- `racket/typed` – the module of Racket with strong typing;
- `scheme` – the backwards compatible module allowing one to use Scheme syntax.

To not overcomplicate the example and to skip any decision process, we will just use `#lang racket`. This will also mean that we will be working in an *untyped* environment, although we will define custom syntax for explicit type annotations and functional types.

2.1 The Grammar and Basic Parsing

First, we specify the language that we will use to make proof terms, which will be a mix of untyped lambda calculus and simple and function types. As such, the BNF grammar of the language is as follows:

$$\begin{aligned}
\texttt{expr} &::= (\lambda x \Longrightarrow \texttt{expr}) \\
&\mid (\texttt{expr}\ \texttt{expr}) \\
&\mid (\texttt{expr} : \texttt{type}) \\
&\mid x \\
\texttt{type} &::= (\texttt{type} \longrightarrow \texttt{type}) \\
&\mid X.
\end{aligned}$$

The reader will recognize, respectively: lambda abstraction, application, type inhabitance (also known as typed variable declaration) and free variables. These are the legal expressions and the types are either simple or functional.

To recognize the structures that appear, we will use the standard Racket structures (records), which will also allow pattern-matching to extract the parts that are needed. This works more or less as a DIY approach to typing, where we make things as verbose as possible, allowing for easy recognition and pattern matching for such types:

```
(struct Lam (var body) #:transparent)          ; lambda abstractions
(struct App (func arg) #:transparent)          ; application
(struct Ann (expr type) #:transparent)         ; explicit type annotations
(struct Arrow (domain range) #:transparent)    ; functional types
```

A little remark about *transparent* vs. *opaque* structures. By default, all structures defined in Racket are opaque. What this means for our purposes is that if one prints such a structure, it doesn't show anything about its internal contents. For example, printing a `Lam` will output `#<Lam>` (see the code examples that follow below). On the other hand, a transparent structure shows its internals when printed, i.e. it will print something like `(Lam 'x 'y)`.

Since by default, all Racket structures are opaque, transparency must be enabled explicitly.

The first goal of the simple proof assistant will be to parse expressions, in order to understand them appropriately and extract the needed parts. For this, we define a function `parse-expr`, which takes a so-called S-expression (standard Racket/Lisp expression) and produces an element that is a legal `expr`.

```
(define (parse-expr s)
  (match s                                ; pattern-match s
    [`(lambda ,(? symbol? x) => ,e)       ; is it a lambda expression?
        (Lam x (parse-expr e))]           ; make it a Lam
    [`(,e0 ,e1)                           ; is it an application?
        (App (parse-expr e0)              ; make it an App
```

```
          (parse-expr e1))]
    [`(,e : ,t)                          ; is it a type annotation (e : t)?
        (Ann (parse-expr e)              ; make it an Ann
             (parse-type t))]
    [`(,e1 ,e2 ,e3 ,r ...)               ; is it a general list?
        (parse-expr
            `((,e1 ,e2) ,e3 ,@r))]       ; parse it by parts
    [(? symbol? x) x]                    ; else, it's a symbol, return it
    [else (error 'parse "bad syntax: ~a" s)]))
```

Notice the very clever use of the quoting and unquoting (including splice) mechanism (detailed in 3.7) in the definition of `parse-expr`, as well as the square bracket delimitation of the matching cases. The quote for the patterns ensures that we are searching for expressions that are either (literally) `(lambda ...)` or `(... ...)`, but inside the quoting we actually want to see what's there, so we evaluate the components using the unquote. The special predicate `(? symbol? x)` is used only in pattern matching, where the first question mark matches anything (similar to `*` in regular expressions). Also note that there is a convention in Lisps to name predicates (i.e. functions that return true or false) ending with a question mark. So basically the last successful case of pattern matching is used for symbols (literals).

Most of the syntax should be clear and it is also accompanied by comments which should ease understanding. The only new piece of syntax which we comment on is the *ellipse* (`...`), which is used to match anything that follows ("the rest").

Similarly, we parse type expressions:

```
(define (parse-type t)
  (match t
    [`(,t1 -> ,t2)                              ; is it a functional type?
        (Arrow (parse-type t1)                  ; make it an Arrow
               (parse-type t2))]
    [`(,t1 -> ,t2 -> ,r ...)                    ; maybe it's a multi-arg function
        (Arrow (parse-type t1)
               (parse-type `(,t1 -> ,@r)))]
    [(? symbol? X) X]                           ; is it a simple type? return it
    [else (error "can't parse this type")]))
```

To make printing clear enough, we define helper functions that do "unparsing", both for expressions and for types:

```
;; pretty-print-expr : expr -> String
(define (pretty-print-expr e)
  (match e
    [(Lam x b) (format "(lambda ~a => ~a)" x (pretty-print-expr b))]
    [(App e1 e2) (format "(~a ~a)"
                         (pretty-print-expr e1)
```

```
                          (pretty-print-expr e2))]
    [(? symbol? x) (format "~a" x)]
    [(Ann e t) (format "(~a : ~a)"
                       (pretty-print-expr e)
                       (pretty-print-expr t))]))

;; pretty-print-type : Type -> String
(define (pretty-print-type t)
  (match t
    [(Arrow t1 t2) (format "(~a -> ~a)"
                           (pretty-print-type t1)
                           (pretty-print-type t2))]
    [else t]))
```

Notice another piece of new syntax in formatted printing, where the `a` placeholder is used, i.e. it will be replaced by the arguments that follow. For example, the syntax `(format "~a" 'x)` will put the symbol `x` in place of `"~a"` when printing.

When evaluating and checking proofs, we will need a *context*, which is defined as a `(listof (List Symbol Type))`. The `listof` keyword is called a *contract* in this case and in short, it *asserts* that what it expects is a list with a symbol and a type. If the context does not respect the contract, we get a specific error. Racket provides extensive support for software contracts, for various items such as structures, functions, variables, but we do not actually need any such complexity here, so we will only indicate [5] for further information. Also, for (judgments in) contexts, we refer the reader to our §1.5, where we see that a context Γ is basically made of lists of typed variable declarations of the form $x : A$.

Now, given the fact that a context is a list of a symbol and a type, we can `pretty-print` it as well using the function:

```
(define (pretty-print-context ctx)
  (cond
    [(empty? ctx) ""]
    [else (string-append (format "\n~a : a"
                                 (first (first ctx))
                                 (pretty-print-type (second (first ctx))))
          (pretty-print-context (rest ctx)))]))
```

2.2 Checking Lambdas

Now, given this setup, we can actually start writing the functions for the proofs. **We remark explicitly that for the purposes of this chapter, we will only present the part that uses lambda terms for proofs.**

First, type-checking, which checks whether an expression has a certain type in a given context.

```
;; type-check : Context Expr Type -> Boolean
;; produces true if expr has type t in context ctx
;; (else, error)
(define (type-check ctx expr type)
  (match expr
    [(Lam x t)              ; is expr a lambda expression?
      (match type           ; then the type must be an arrow
        [(Arrow tt tw) (type-check (cons `(,x ,tt) ctx) t tw)]
        [else (cannot-check ctx expr type)])]
  [else (if (equal? (type-infer ctx expr) type) true    ; fail for other types
            (cannot-check ctx expr type))]))

;; the error function
(define (cannot-check ctx expr type)
  (error 'type-check "cannot typecheck ~a as ~a in context:\n~a"
    (pretty-print-expr expr)
    (pretty-print-type type)
    (pretty-print-context ctx)))
```

Then the function that tries to make a type inference.

```
;; type-infer : Context Expr -> Type
;; tries to produce type of expr in context ctx
;; (else, error)
(define (type-infer ctx expr)
  (match expr
    [(Lam _ _) (cannot-infer ctx expr)]     ; lambdas are handled in type-check
    [(Ann e t) (type-check ctx e t) t]      ; check type annotations
    [(App f a)                              ; function application
        (define tf (type-infer ctx f))
          (match tf                         ; must be arrow type
            [(Arrow tt tw) #:when (type-check ctx a tt) tw] ; when the rest typechecks
            [else (cannot-infer ctx expr)])]
    [(? symbol? x)                          ; for symbols
       (cond
         [(assoc x ctx) => second]          ; if it's a list, the second is the type
         [else (cannot-infer ctx expr)]))]) ; else, not okay

;; the error function
(define (cannot-infer ctx expr)
  (error 'type-infer "cannot infer type of ~a in context:\n~a"
    (pretty-print-expr expr)
    (pretty-print-context ctx)))
```

2.3 Basic Testing

Now we can define a function that checks some basic proofs like so:

```
(define (check-proof p)
    (type-infer empty (parse-expr p)) true)
```

This way, if errors do not appear, the function will successfully apply the `type-infer` procedure and return `true`, which can be seen as a "successful exit code".

Using this, we can use the Racket testing module to check some basic proofs, such as:

```
(require test-engine/racket-tests)              ; import the test module

;; check whether check-proof returns true => good proof
;; lambda xy.x : A -> (B -> A)
(check-expect
  (check-proof '((lambda x => (lambda y => x)) : (A -> (B -> A))))
    true)

;; lambda xy.yx : (A -> ((A -> B) -> B))
(check-expect
  (check-proof '((lambda x => (lambda y => (y x))) : (A -> ((A -> B) -> B))))
    true)

;; lambda fgx.f(gx) : ((B -> C) -> ((A -> B) -> (A -> C)))
(check-expect
  (check-proof '((lambda f => (lambda g => (lambda x => f (g x)))) :
                    ((B -> C -> ((A -> B) -> (A -> C))))))
    true)

(test)          ;; => All 3 tests passed.
```

Formally, what `check-proof` does is to verify well-formedness of typing judgments for lambda expressions (see our §1). As such, for example, the first proof checks the typing for the Church Boolean `true` in our §1.4.

The other two proofs are just simple exercises. We can also write, for example, the typing for the Church numeral c_1 (again, see §1.4). Given that:

$$c_1 = \lambda s.\lambda z.sz = \lambda sz.sz,$$

we have:

```
;; c1 = lambda sz . sz : N -> N
(check-expect
  (check-proof '((lambda s => (lambda z => (s z))) : (N -> N))))
(test)                  ;; => All 1 test passed.
```

CHAPTER 3

PIE

Pie is a language implemented in Racket meant to facilitate programming and theorem proving with dependent types. It is best described in the book [7], which is written in a very original way, by means of dialogue, as a theater play, with emphasis on interaction and asking questions. In fact, the book is only one in a series generally titled *The Little X*, where *X* so far has been *Schemer, Lisper, Prover, Haskeller*.

In this chapter, we will present some basic syntax and features of Pie first and then insist on its implementation of dependent types. Most of the presentation will follow [7].

3.1 Basic Syntax and Features

Since Pie is written in Racket, its syntax is similar to Racket's. Moreover, the language is available as a Racket module, hence it can be installed using the Racket packet manager, `raco`. After that, any source code file will start with the pragma (module call) `#lang pie` and it will be interpreted accordingly.

Since the language is strongly typed, that is, the user must associate a type to each variable or function they use, all the definitions must be preceded either by a `claim` or a `the` statement.

A `claim` statement declares the type of a function or variable and must precede any definition, whereas a `the` expression does a type annotation for variables. Simple examples follow:

```
#lang pie

(the (Pair Atom Atom)
     (cons 'spinach 'cauliflower))
;; declares that the expression (cons 'spinach 'cauliflower)
;; makes a Pair of Atoms. more on these below
```

```
(claim one Nat)                 ; one is of type Nat
(define one (add1 zero))        ; zero is primitive
(the Nat (add1 one))            ; 2 is a Nat
```

Two basic types that can be used are `Nat`, for positive integers, `Atom` for quoted variables (symbols). Then, we can use *constructors* and *type constructors*. An expression that is preceded by a constructor is called a *value*. For example, one constructor of the type `Nat` is `zero`, which returns `0` and another one is `add1`, which is basically the successor function. So for example, we can obtain a value like so:

```
#lang pie

(claim two Nat)
(define two
  (add1
    (add1 zero)))
```

Also, atoms are values, so the type `Atom` does not have a constructor:

```
#lang pie

(claim dog Atom)
(define dog 'dog)
```

Then, further, we can build types using *type constructors*. One example is the type constructor `Pair` which makes a pair of elements having (possibly) different types. A constructor for such a type is `cons`:

```
#lang pie

(claim myDogs (Pair Atom Atom))
(define myDogs (cons 'Ricky 'Rocky))
```

We will meet more types and (type) constructors further, so we will explain them on the spot. Notice that we can also use lambda expressions, as throughout Racket.

One last important feature which we mention here is that the type theory that we use is, in a sense, *closed* or *total*. That is, anything must have a type... *including types!* Without going into details, we mention that the type of types is one all-encompassing *universe*, denoted mathematically by $\mathcal{U}$ and in Pie, by `U`. We will meet universes when we build Π-types in Pie.

3.2 Induction and Dependent Types

We now are heading towards more complex features of Pie, of which dependent types are an essential example. But before we get there, we will introduce some functions which work on the

basis of induction. We provide explicit examples instead of focusing on theoretical aspects. The theory between the principle of mathematical induction was introduced on the type of natural numbers in §1.8.

As such, we first introduce the Pie function `which-Nat`. The general syntax is:

```
(which-Nat  target  base  step),
```

which is a function that acts on `Nat` variables. It works as follows:

- it first checks whether `target` is `zero`;
- if it is, the function returns the value of `base`;
- otherwise, `target` must be some `(add1 n)`, for some `n`, since these are the only kinds of values of type `Nat`. In this case, it returns the value of `(step n)`.

This pattern will be encountered in some other functions as well, so we will spend a little on explaining it better. The first argument (both for `which-Nat` and some other similar functions that we meet is, in a sense, "the main argument", since depending on its form, the rest happens. In the case of `Nat` values, we can only encounter either `zero` or `add1 n`. In the first case, the function returns `base`, as if we are in the base case of an inductive proof. In the second case, notice that if `target` has the form `add1 n`, then we have access to the *preceding* case, `n`. So we also have `n` and `n + 1 = (add1 n)`. The function now offers the flexibility to go not just from `n` to `n + 1`, but from `n` to `(step n)`. By abuse of language, we will still call this *the inductive step*. As such, whenever we use `which-Nat`, as well as the other similar functions that will appear, we want to make sure we know what to do in *the base case* and in the *inductive step*. In other words, we must have useful definitions both for `base` and for `step`.

Some simple examples with `which-Nat` follow below. Notice that since `which-Nat` is a standard library function for Pie, we don't have to `claim` it.

```
#lang pie

(which-Nat zero
           'naught
          (lambda (n) 'more))           ;; => 'naught
(which-Nat 4
           'naught
           (lambda (n) 'more))          ;; => 'more
(which-Nat 5
           0
           (lambda (n) (+ 6 n)))        ;; => 10
```

The first case is self explanatory: the `target` is exactly `zero`, so we just get the base case, which is `'naught`.

In the second case, the `target` is not `zero`, so we will evaluate the `step` case. For this, we first notice that `4 = (add1 3)`, so `n = 3` and in this case, we get the value of `((lambda (n) 'more) 3)`, which is `'more`. Notice that in this lambda expression, the evaluation at `n = 3` does nothing, since the lambda associates `'more` to any argument.

In the third case, however, we do have a more meaningful lambda. Again, `target` is not `zero`, so we get the `step` case. Since `5 = (add1 4)`, we get:

```
((lambda (n) (+ 6 n)) 4) = 10.
```

3.3 Example: `Pair` Types and functions

Let us now define a fully custom type, along with its eliminator and a function that acts on values of this type. Following [7], we will call the type `Pear`. This is actually a custom pair of two natural numbers so in a way, we are redefining the type `(Pair Nat Nat)`. We first present the example and comment it afterwards:

```
#lang pie

(claim Pear U)                      ; claim the custom type (of universal type)
(define Pear (Pair Nat Nat))        ; a Pear is a Pair of two Nats

(claim Pear-maker U)                ; type constructor, with the universal type
(define Pear-maker                  ; actual definition of the type constructor
  (-> Nat Nat Pear))                ; a functional type from 2 Nats to a Pear

(claim elim-Pear                    ; the type eliminator
    (-> Pear Pear-maker Pear))      ; takes a (Pear, Pear-maker), gives a Pear
(define elim-Pear                   ; the definition of the eliminator
  (lambda (pear maker)              ; is a lambda taking a pear and a maker
    (maker (car pear) (cdr pear)))) ; sends the maker on the first part of pear
                                    ; and on the second part of pear

(claim pearwise+                    ; addition for Pears
  (-> Pear Pear Pear))              ; takes two Pears and returns a third one
(define pearwise+                   ; (a, b) + (c, d) = (a + c, b + d)
  (lambda (x y)                     ; take the two parts (Nats)
    (elim-Pear x                    ; split the first    (Nat)
      (lambda (a1 d1)               ; into a1 and d1     (Nats)
        (elim-Pear y                ; split the second   (Nat)
          (lambda (a2 d2)           ; into a2 and d2     (Nats)
            (cons                   ; use the Pair constructor
              (+ a1 a2)             ; to make a Pair of their added components
              (+ d1 d2)))))))))
```

Since all the definitions are actually lambda expressions, it could seem difficult to test some examples. Here is a use case, which we will comment afterwards:

```
(elim-Pear (cons 3 17)
           (lambda (a d) (cons d a)))
;; is the same as
((lambda (pear maker)
    (maker (car pear) (cdr pear)))
  (cons 3 17)
  (lambda (a d) (cons d a)))
;; which finally reduces to
(cons 17 3)
```

The explanations are something like this. Note that in order to correctly evaluate `elim-Pear`, its arguments must be a `Pear` (that is, a "custom" `Pair`), which we must write using a `cons`, to make sure the output has the required type. Otherwise, for example, just using `(3 17)` or some other seemingly clear pair would not do. The second argument must be a `Pear-maker`, which we can see in the code that it is a function that takes two `Nat`s and returns a `Pear`. In our case, it is a lambda expression with two arguments which uses `cons` to make sure it returns a `Pear`.

Then we actually evaluate the lambda expression which is equivalent to the `elim-Pear` expression. The outermost lambda uses two arguments, `pear` and `maker`, so when evaluating, they bind as:

- `pear` becomes `(cons 3 17)`;
- `maker` becomes `(lambda (a d) (cons d a))`, which is just a flip function for `Pair`s (or `Pear`s).

Looking at the outermost lambda again, we see that what it does to its arguments is to apply the `maker` to the halves of the `pear`. In our case, taking the `car` of `pear` returns `3` and its `cdr` is `17`. So we just `cons` them in reverse order and get `(17 3)`. In fact, the answer of the REPL is:

```
(the (Pair Nat Nat) (cons 17 3))
```

which shows the types explicitly.

To make matters more difficult, but also more verbose and more in the DIY style we are taking here, if one tries to use the `pearwise+` sum, one finds surprisingly that the `+` function is not defined! Therefore, addition is not a part of the standard library of Pie. This makes sense in a way, since using dependent types enforces custom definitions either in a uniform way using `Pi` or special definitions for each type. In this case, in order to define addition for the type `Nat`, which we need for the components of a `Pear = (Pair Nat Nat)` type, we also require another function which behaves somewhat inductively. This is `iter-Nat`, whose general syntax is similar to what we will be seeing more of in §3.2 and §3.4. That is, we have:

```
(iter-Nat target base step)
```

which works as follows:

- if the value of `target` is `zero`, it returns the value of `base`;
- otherwise, if `target` is some `(add1 n)`, the value of the expression is:

  `(step (iter-Nat n base step))`.

  In other words, it performs the "inductive processing step" on the previous case.

Having this at hand, we can define addition for the `Nat` type.

```
#lang pie

;; first, define the inductive step
(claim step+ (-> Nat Nat))              ; it processes one Nat and returns another
(define step+                           ; just add one,
  (lambda (+n-1)                        ; but we renamed the argument to +n-1
    (add1 +n-1)))                       ; we are using the previous case

;; now the addition
(claim + (-> Nat Nat Nat))              ; addition for the type Nat
(define +
  (lambda (n j)
    (iter-Nat n j step+)))
```

Figure 3.1: Addition for the `Nat` type

Two remarks are in order at this point. First, a general Racket remark, which in fact applies to most Lisps: variable names can use whatever symbols the user wishes, including `+, -, /, ?` etc. In fact, there are even common practices with them, for example, calling a predicate (a function that returns a Boolean) with a question mark at the end. We have met this in §2 for symbol parsing with `symbol?`.

The second remark is about the actual definition of `+`. Notice that it is a two-argument function which just calls `iter-Nat`. By definition, if the first argument `n` is `zero`, we just get `j` (the second argument), since $0 + j = j$. Otherwise, we use the "inductive step" `step+` to keep adding `1`. Notice, moreover, that by the definition of `iter-Nat`, whenever we use the `step` case, the `target` argument decreases. So it means that `step+` will be called exactly `n` times, for `n, n-1,` down to `1`.

We should also add the remark that no other `Nat` is defined except for `zero` by default. So any addition we want to test must be preceded by the `claim` and the definition of the numbers, such as:

```
#lang pie

(claim three Nat)
(define three
  (add1 (add1 (add1 zero))))

;; now test addition
(+ (add1 zero) three)           ; => (the Nat 4)
```

This was an exercise in the definition of a custom type, along with its maker, eliminator and a function among variables of this type. We now focus on standard types and examine how functions on types obtained using the `Pair` type constructor are defined using dependent types. What we will be writing is already included in the standard library, but their breakdown and re-implementation is useful for education purposes.

The basic idea is the following: as we mentioned in the section on dependent types (§1.5), want we want to achieve is to be able to define functions on variables having different types *in the same way*. Here is how some functions on `Pairs` can be implemented:

```
#lang pie

(claim elim-Pair                ; "consumer" (eliminator) for Pairs
  (Pi ((A U) (D U) (X U))       ; will be used throughout, since it's
    (-> (Pair A D)              ; defined on product (parametrized) types
        (-> A D X) X)))
(define elim-Pair
  (lambda (A D X)
    (lambda (p f)
      (f (car p) (cdr p)))))

(claim kar                      ; replacement for car
  (-> (Pair Nat Nat) Nat))      ; for pairs of Nats
(define kar
  (lambda (p)
    (elim-Pair Nat Nat Nat p
      (lambda (a d) a))))

(claim kdr                      ; replacement for cdr
  (-> (Pair Nat Nat Nat)))      ; for pairs of Nats
(define kdr
  (lambda (p)
    (elim-Pair Nat Nat Nat p
      (lambda (a d) d))))
```

```
;; flipping two elements of a Pair
(claim flip
  (Pi ((A U) (D U))
    (-> (Pair A D) (Pair D A))))
(define flip
  (lambda (A D)                 ; abstract types first
    (lambda (p)                 ; then the actual pair
      (cons (cdr p) (car p))))) ; flip its halves
```

We now explain the code above, since it contains some aspects that have not been met so far. First, notice that we define the eliminator for `Pair` types[1], but in such a way that it uses dependent types. This way, we use the same definition regardless of the actual types that will appear in the arguments by a supplementary abstraction on types.

Therefore, we first need to write that `elim-Pair` will be a `Pi` type that depends on three types `A : U, D : U, X : U`. Then, what it will actually do is to provide a function whose first argument is of type `Pair A D` and the second argument is another function of type `A -> D -> X` and finally, the output will have type `X`. The action will become clear in the definition that follows the `claim`. As such, how this works is the following:

- it first abstracts with respect to types. This is a key step in a dependent type, since as we mentioned, it must work on various types of arguments that it is fed and this is the place where the types of those arguments are made explicit;
- then, the action is another abstraction, this time on two arguments, one being a function `f` and a pair `p`. How it works is that it splits the `p` in two, using `car` and `cdr` and it applies the function to both of these halves. Its use will become even more transparent in the examples that follow.

Then we implement `kar`, which mimics the behaviour of `car` for `(Pair Nat Nat)` arguments, i.e. it returns the first part. After `claim`ing its type, which is precisely `(Pair Nat Nat) -> Nat`, we `define` it as follows:

- we use `elim-Pair`, since it is the only "consumer" of `Pair` types;
- the first arguments of `elim-Pair` are the actual types with respect to which we do the first abstraction (i.e. the types that enter in `Pi`). In this case, all three of them are `Nat`;
- lastly, the non-type argument `p`, which is exactly the pair of naturals. How this works is to provide a function which, given two arguments, it returns the first one.

[1] Formally, we should say something like `(Pair X Y)`, where `X` and `Y` are types, since `Pair` itself is just a *type constructor* that takes two arguments which are types to produce a type. But with this remark, we acknowledge the language abuse which we've done and continue doing it.

Let's do a bit of "pattern-matching" in this definition, namely let's match the concrete arguments in `kar`'s definition with the generic arguments in the definition of `elim-Pair`. First of all, as we mentioned, `A = D = X = Nat`. Then, the second argument of `elim-Pair` must be of type `(Pair Nat Nat) -> (Nat -> Nat -> Nat) -> Nat`. That is, it must accept a pair of `Nat`s and a function of two `Nat` arguments that produces a third `Nat` and finally it must return a `Nat`. But this becomes irrelevant in this case, since in `kar` we are abstracting by some generic `p` and make it act like a `lambda` that returns the first half of its arguments. That is, what `kar` is doing (after mentioning the types) is the assignment:

$$\texttt{p} \mapsto ((a, d) \mapsto a)\,.$$

It is as if `p` gets renamed to `first`.

Similarly, `kdr` is a sort of `second`.

A very similar example was the custom `elim-Pear` function we defined and used in §3.3.

Now to test the `flip` function, we need to call it accordingly. For example, we can first make it a particular flip function for certain types:

```
(flip Nat Atom)
```

is just the function that requires a `(Pair Nat Atom)` argument which it binds to `p` and is then:

```
(lambda (p) (cons (cdr p) (car p))).
```

A particular example is:

```
((flip Nat Atom) (cons 17 'apple)),
```

which replies with:

```
(the (Pair Atom Nat) (cons 'apple 17)).
```

3.4 Example: `List` Types and Functions

A sort of generalization of `Pair` types are *lists*. We now show how some functions on `List` types[2] can be implemented using dependent types.

First, the prerequisites. `(List X)` is a type, where `X` is a type and this type has two *constructors*:

- `nil`, which makes an empty list (similar to `zero` for `Nat`);

[2] Again, as in the case of `Pair`, `List` is a *type constructor*, so formally it is not correct to say "`List` type". It should be something like `List X` type, where `X` is a type. But as in the case of `Pair`s, we acknowledge and keep using this expression.

- `::`, which is the equivalent of `cons` for pairs.

Also as in the case of `Nat`, we have a special "inductive" function `rec-list`. Its general syntax is:

```
(rec-List target base step)
```

and this produces an element of type `X` when:

- `target` is of type `(List E)`;
- `base` is of type `X`;
- `step` is of type `(-> E (List E) X X)`.

How this works is similar to `which-Nat` with a twist:

- if `target` is `nil`, it returns the value of `base`;
- if `target` is some `(::  elt rest)`, where `elt` is a variable and `rest` is a list, it recursively calls itself with `target` replaced by `rest`.

Here is a use case, where we redefine the length function for a list. We will use again `Pi` types, since we want it to work for lists of elements of any type.

```
#lang pie

(claim length                        ; length of a list
  (Pi ((E U))                        ; using dependent types
    (-> (List E) Nat)))              ; takes a list and returns a natural
(define length
  (lambda (E)                        ; abstract the type
    (lambda (es)                     ; abstract the argument
      (rec-List es                   ; make the argument to be the target
                0
                (step-length E)))))  ; we need the "inductive step"

;; "inductive step" for computing length of a list
(claim step-length
  (Pi ((E U))                        ; a dependent function
    (-> E (List E) Nat Nat)))        ; takes a type, a list of elements of that type,
                                     ; a Nat and returns a Nat
(define step-length
  (lambda (E)                        ; abstract the parameter type
    (lambda (e es length-es)         ; the rest of the arguments
      (add1 length-es))))
```

```
;; special version for Atom
(claim length-Atom
  (-> (List Atom) Nat))
(define length-Atom
  (length Atom))
```

Since `length` abstracts on the type first and then the list, a test should be written like:

```
(length Nat (::  1 (::  2 (::  3 nil))))  ; => (the Nat 3)
```

We spend some space here to detail the `rec-List` function, but before doing that, we go a little back to its simpler version, `rec-Nat`, for arguments of type `Nat`. The general syntax follows the pattern we've seen so far:

```
(rec-Nat target base step)
```

and also as before, when the value of `target` is `zero`, we get the value of `base`. For the "inductive step", assume the value of `target` is some `(add1 n)`. Then `step` must be a function of two arguments, of which the first will be bound to `n` and the second will be bound to a recursive call of the same `rec-Nat` expression, with `target` replaced by `n`.

Example:

```
(rec-Nat
  (add1 zero)                   ; target, so n = zero, but target = 1
  0                             ; base
  (lambda (n-1 almost)          ; step (two-argument function)
    (add1 (add1 almost))))

;; first, using rec-Nat definition, reduces to:
((lambda (n-1 almost)
    (add1 (add1 almost)))
  zero
  (rec-Nat zero                 ; recursive call with target = 0
    0                           ; same base case
    (lambda (n-1 almost)        ; same step function
      (add1 (add1 almost)))))

;; then using lambda rules of computation reduces to:
(add1 (add1
        (rec-Nat zero
                 0
                 (lambda (n-1 almost)
                    (add1 (add1 almost))))))

;; and finally
(add1 (add1 0))                 ; => (the Nat 2) provided 2 is claimed and defined!
```

A very simple and interesting application of `rec-Nat` is to define the function for a Gauss sum. When calling `(gauss n)`, it should return the sum of all the integers up to (including) n, which is $\frac{n(n-1)}{2}$. We will be using addition for the `Nat` type, defined in 3.1.

```
#lang pie

;; the inductive step
(claim step-gauss
  (-> Nat Nat Nat))
(define step-gauss
  (lambda (n-1 gauss-n-1)
    (+ (add1 n-1) gauss-n-1)))

;; now the Gauss function
(claim gauss
  (-> Nat Nat))
(define gauss
  (lambda (n)
    (rec-Nat n
      0
      step-gauss)))
```

Now the explanation for `rec-List` is similar to the `rec-Nat` case, with the appropriate "translations":

- `zero` is replaced by `nil`;
- `add1` is `::`.

For typing, it follows that the eliminator `(rec-List target base step)` has type `X` when:

- `target` has type `(List E)` (since it will be "decomposed");
- `base` has type `X` (since it gets returned in the base case);
- `step` has type `(-> E (List E) X X)`, which should be parsed as `(E, (List E) -> X) -> X`. This is because `step` should be a two-argument function, first of which being the `List E` of `target` with an element stripped off and the second argument should be the entire `rec-List` expression, called with the modified `target`.

Now the innards of `length` should be transparent: it uses `rec-List` to increase the length of the list by one, while popping out elements from it (the `target` argument).

3.5 More on Product Types

This section and its subsequents will be heavily based on the first chapter of [10] and the idea is to insist on some theoretical aspects of certain composite types, after which we will see if and how they can be implemented in Pie. There are some intentional design decisions that impose certain limits on Pie and we will see how they can be eluded.

First, assume we have two types $A, B : \mathcal{U}$ and we want to introduce their product type $A \times B : \mathcal{U}$, which is called their *(Cartesian) product type.* In the particular case when make the void choice both for A and B, we get the *unit type*, also called the *nullary product type*, denoted by $\mathbf{1} : \mathcal{U}$.

The canonical elements of type $A \times B$ will be pairs of the form (a, b), where $a : A$ and $b : B$, whereas the unique element of type $\mathbf{1}$ is denoted by some generic $* : \mathbf{1}$.

Note that if in set theory, both functions and (ordered) pairs are constructed, in type theory they are taken as primitive concepts, both of them.

Hence, we have above, in MLTT parlance, the *introduction rules*: $A \times B$ type, for A type and B type. Or, using universes: $A \times B : \mathcal{U}$, for $A, B : \mathcal{U}$ and similarly for the unit type $* : \mathbf{1}$ (for the void choice).

For the constructor of this type, we make use of the *uniqueness principle* (which can be proved using equality types and for which we refer the reader to [10, p. 29]), which states that all elements of type $A \times B$ are ordered pairs. So if we know how to construct ordered pairs, we can inhabit product types. But there is nothing to prove, since we assumed that pairs are primitive concepts. That is, starting with $a : A$ and $b : B$, we can directly construct $(a, b) : A \times B$.

Now we need the *type eliminators*, which show us how to use ordered pairs. For this purpose, we want to construct functions out of product types, i.e. functions which take pairs as arguments. We will do this by relying on previously introduced types, namely function types (with their constructor, λ). So, we start with a function $g : A \to B \to C$, or, in other words, with the function type $A \to B \to C$, inhabited by some g. This type can be parsed as $(A \to B) \to C$, so g is some $\lambda a.\lambda b.\Phi$[3], where Φ is a functional expression that may or may not contain a and b.[4]

Using this g, we want to construct $f : A \times B \to C$. This is easy and we have the *elimination rule* for product types which is the definitional equality below:

$$f((a, b)) :\equiv g(a)(b).$$

There are two particular cases of interest, namely the *canonical projection* functions:

$$p_1 : (a, b) \mapsto a, \quad p_2 : (a, b) \mapsto b.$$

And furthermore, they can be put in a more general framework using a function that is called

[4]This explicit, iterated lambda expression is sometimes called the *uncurried* version for a two-argument function. There is also the *curried* version, written as $\lambda ab.\Phi$, which is explicit enough.

[4]We should have been more explicit by using *typed* lambda expressions, i.e. $\lambda(a : A).\lambda(b : B).\Phi$, but we omitted the types, hoping they are understood from the context.

a *recursor* for product types:

$$\mathrm{rec}_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to A \times B \to C$$
$$\mathrm{rec}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b).$$

This function is parametrized by some type $C : \mathcal{U}$ and has two arguments: one is a two-valued *dependent function* $A \to B \to C$ and the other is a pair of type $A \times B$. Using this, we can recover the projection functions by the following definitional equalities:

$$p_1 :\equiv \mathrm{rec}_{A\times B}(A, \lambda ab.a)$$
$$p_2 :\equiv \mathrm{rec}_{A\times B}(B, \lambda ab.b).$$

For the unit type, this is trivial:

$$\mathrm{rec}_{\mathbf{1}} : \prod_{C:\mathcal{U}} C \to \mathbf{1} \to C, \quad \mathrm{rec}_{\mathbf{1}}(C, c, *) :\equiv c.$$

Now onto the Pie implementation. We have already met the `Pair` type constructor, which can be seen as the Cartesian product. That is, if `A` and `B` are types, `(Pair A B)` is a product type. We already know which are its eliminators, namely `car` and `cdr`, which are, respectively, the first and second projection. Moreover, its constructor is `cons`.

Therefore, we continue by adding the unit type and the recursor. The unit type is built-in Pie and it is called `Trivial`. Its only element (or constructor) is called `sole`. Hence, the expression `(the Trivial sole)` does the typing $* : \mathbf{1}$.

As for the recursor, there are two particular shortcomings of Pie, which are actually intentional design decisions[5], but they can be overcame.

First, there are no "generic types" in Pie, so there is no equivalent of $A : \mathcal{U}$, so we have to use particular types that are built-in the language, such as `Nat` or `Atom`.

Second, the `Pi` type constructor for dependent types must bind all types that appear in its `claim`. That is, we will not be able to make explicit the fact that the recursor $\mathrm{rec}_{A\times B}$ uses some *fixed* types A and B and the only type parameter is C.

Those being said, an implementation can be as follows:

```
#lang pie

(claim recAxB
       (Pi ((A U) (B U) (C U))              ; as if all are type params
           (-> (-> A B C) (Pair A B) C)))
(define recAxB
  (lambda (A B C)                           ; abstract (all) the types first
    (lambda (g p)                           ; g is (lambda ab.F) and p is a Pair
      ((g (car p)) (cdr p)))))
```

[5] Thank you, David Thrane Christiansen for explaining them to me!

And now for example, we can test as follows:

```
((recAxB Nat Nat Nat)
    (lambda (x y) (add1 y)) (cons 2 3))     ; => (the Nat 4)
;;  |------ g ------------| |--- p ----|
```

We can also define the first projection for the `Nat` type using the recursor by:

```
(claim p1 (-> (Pair Nat Nat) Nat))
(define p1
  ((recAxB Nat Nat Nat)
     (lambda (x y) x)))

;; test
(p1 (cons 2 3))                             ; => (the Nat 2)
```

For the particular case of the unit type, we have:

```
(claim recU
  (Pi ((C U)) (-> C Trivial C)))
(define recU
  (lambda (C)
    (lambda (c d) c)))                      ; d can only be sole

;; test
((recU Trivial) sole sole)                  ; => (the Trivial sole)
((recU Nat) zero sole)                      ; => (the Nat 0)
```

3.6 Coproduct (Sum) Types

Starting with two types $A, B : \mathcal{U}$, we can form their *coproduct* or *sum* type, denoted by $A + B : \mathcal{U}$. This corresponds to disjoint unions in set theory, i.e. unions in which one can always identify precisely from where are the elements taken, either from the first term or from the second[6].

The nullary version, where we don't choose neither A nor B is the *null type*, which doesn't distinguish its elements and which we denote by $\mathbf{0} : \mathcal{U}$.

Having said this, it is clear that to construct elements of type $A + B$, we need the so-called *left (right) injections*, which we denote by $\mathrm{inl}(a) : A + B$ for some $a : A$ and $\mathrm{inr}(b) : A + B$ for some $b : B$.

[6]From a logical point of view and taking into consideration the connection between types and propositions, coproduct types correspond to *logical (exclusive) disjunctions*. That is, the type $A + B$, when A and B are interpreted as propositions, is meant to mean *A xor B*. In words, this is "*Either A or B*", which is reflected best in the Haskell syntax `data Either a b = Left a | Right b`.

Now, eliminators for this type, i.e. functions $f : A + B \longrightarrow C$ are constructed by components. That is, we need functions $g_l : A \longrightarrow C$ and $g_r : B \longrightarrow C$ and then define:

$$f(\text{inl}(a)) :\equiv g_l(a), \quad f(\text{inr}(b)) :\equiv g_r(b)$$

and invoke a *uniqueness principle* for coproduct types, which states that all elements of such type are left (or right) injections. This means that the definition above is by *exhaustive case analysis.*

As in the case of product types, we have a recursor for sum types, namely:

$$\begin{aligned}
&\text{rec}_{A+B} : \prod_{C:\mathcal{U}} (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow A + B \longrightarrow C \\
&\text{rec}_{A+B}(C, g_l, g_r, \text{inl}(a)) :\equiv g_l(a) \\
&\text{rec}_{A+B}(C, g_l, g_r, \text{inr}(b)) :\equiv g_r(b).
\end{aligned}$$

Since there are no elements of type $\mathbf{0}$, we trivially construct a function $f : \mathbf{0} \longrightarrow C$ by providing no equations. So basically the recursor is:

$$\text{rec}_{\mathbf{0}} : \prod_{C:\mathcal{U}} \mathbf{0} \longrightarrow C,$$

which constructs a canonical element of type C starting from nothing. This is the type theoretical expression of the logical principle *ex falso quodlibet*, which states that a false statement can imply anything. That is, starting with a false assumption, we can derive anything.

Now onto the Pie implementation. Conveniently enough, Pie already contains the coproduct types, whose type constructor is `Either`. That is, if `A` and `B` are types, then `(Either A B)` is their sum type written mathematically as $A + B$. Its constructors are `left` and `right`, which means that `a` is of type `A`, then `(left a)` has type `(Either A B)` and if `b` is of type `B`, then `(right b)` has type `(Either A B)`.

Also, `Absurd` is the empty type.

However, for case analysis, we need another important theoretical item. This is what Conor McBride called a *motive* in [15] and informally, it is a parametrized type choice. That is, it is a function mot $: A \longrightarrow \mathcal{U}$, where A is some available type. Although this concept is worth discussing at length, we will restrict ourselves to an understanding based on some examples. Aside from the `rec-` and `which-` functions that were used to construct (more or less) inductively certain elements, there is another family of such functions, prefixed by `ind-`. We will explain it first in the case of the `Nat` type, as it is the most familiar background for inductive definitions. As such, the general syntax of the `ind-Nat` function is:

```
(ind-Nat target mot base step),
```
(3.1)

which deserves further explanation. As stated above, `mot` is a parametrized type choice, which in the case of `ind-Nat` means `mot` is `(-> Nat U)`. This allows to change the type for each `Nat`

argument or, in other words, to have a *sequence of types* (indexed by natural numbers). The use of such a feature is evident in the case of polymorphic lists, for example.

Further, `base` has type `(mot zero)`, thus preserving the intuitive meaning of "base case" and `step` has a type of the form:

```
(Pi ((n-1 Nat)) (-> (mot n-1) (mot (add1 n-1)))),
```

again preserving the intuitive meaning of "inductive step". In words, `step` has a type that is dependent of the previous case and it is actually a function which relates the previous type `(mot n-1)` with the current type, `(mot (add1 n-1))` (`(add1 n-1)` is actually `n`).

Finally, the whole `ind-Nat` expression returns a type that is `(mot target)`.

We can detail further, knowing that `target` can have only two values, `zero` or `(add1 n)`:

- if `target` is `zero`, then the whole `ind-Nat` expression from equation (3.1) has type `(mot zero)` and it coincides with `base`;
- if `target` is `(add1 n)`, then the whole `ind-Nat` expression from equation (3.1) has type `(mot (add1 n))` and it coincides with:

```
(step n (ind-Nat n mot base step)).
```

This `ind-` function comes in handy when discussing the simpler case of `Either` types. For this case, the respective `ind-Either` function has a simpler syntax:

```
(ind-Either target mot base-left base-right).          (3.2)
```

As in the case of `ind-Nat`, the motive `mot` means an appropriate parametrized choice of type. Therefore, in this case, `mot` has type `(-> (Either L R) U)`. Then, `base-left` shows what type is chosen for the `left` part of the `Either` type. This means that the type of `base-left` is:

```
(Pi ((x L)) (mot (left x)))
```

and similarly for `base-right`.

Note that unlike the `ind-Nat` function, we don't have a `step` for `ind-Either`, since there are no steps when constructing an `Either` type; there are only two cases, so the target can be either some `(left x)` or some `(right y)`.

The typing rule for this function states that, under the typing assumptions above, the expression from equation (3.2) has type `(mot target)`.

How this works for case analysis is explained in the computation rules:

- `(ind-Either (left x) mot base-left base-right)` is an expression with type `(mot (left x))` which is computed via `(base-left x)`;

- `(ind-Either (right y) mot base-left base-right)` is an expression with type `(mot (right y))` which is computed using `(base-right y)`.

Now we can adapt the recursor for product types in order to get the version for sum types:

```
#lang pie

(claim recA+B
    (Pi ((A U) (B U) (C U))
        (-> (-> A C) (-> B C) (Either A B) C)))

(define recA+B
  (lambda (A B C)
    (lambda (gl gr tgt)
      (ind-Either tgt
        (lambda (_) C)
        (lambda (_) (gl _))
        (lambda (_) (gr _))))))
```

How this works is explained below:

- we are abstracting over the types first, as it is needed in the `Pi`, hence the first `(lambda A B C)`;
- then, the arguments are `gl : A -> C`, `gr : B -> C` and `tgt : (Either A B)`, so `tgt` is some `(left x)` or `(right y)`;
- using the `Either`-eliminator `ind-Either`, we have the motive which always returns something of type `C`;
- next, the `base-left` case corresponds to when `tgt` is some `(left x)`. For that case, we want to apply `gl` to `x`;
- otherwise, `tgt` is `(right y)`, so we apply `gr` to `y`.

Notice how the placeholder syntax `(_)` is very useful in this case, as it automatically knows what to do. Namely, for example, for the `left x` case, we want to apply `gl` to `x`, discarding the `left`, which is automatically done using the placeholder.

This could have worked equally well with explicit syntax, such as:

```
(define recA+B
  (lambda (A B C)
    (lambda (gl gr tgt)
      (ind-Either tgt
        (lambda (mot) C)
        (lambda (x) (gl x))
        (lambda (y) (gr y))))))
```

but one can argue that it could be confusing since we know that the last two `lambdas` must correspond to some `left` or `right` cases. So it's probably better to leave the arguments hidden.

We can now test for the `Nat` type like so, for example:

```
((recA+B Nat Nat Nat)           ; A = B = C = Nat
     (lambda (x) (add1 x))      ; gl = +1
     (lambda (y) zero)          ; gr = 0
     (left 3))                  ; => (the Nat 4)
((recA+B Nat Nat Nat)
     (lambda (x) (add1 x))
     (lambda (y) zero)
     (right 3))                 ; => (the Nat 0)
```

3.7 Example: The Type of Booleans

Pie does not contain the type of Boolean variables, but this can be implemented easily. Moreover, we will follow the presentation from [10, §1.8] and show how this type can be defined using the notions that we have met so far.

First, it is clear that the type of Booleans can be defined as a coproduct type. Intuitively, this means that we are using the *excluded middle* principle and we think that any proposition (resp. type) is either true (resp. inhabited) or false (resp. uninhabited). This means that the type of Booleans, which we denote by `Bool` or $\mathbf{2}$ can be seen as the coproduct of two unit types, $\mathbf{2} = \mathbf{1} + \mathbf{1}$.

Although seemingly simple, this type is of big theoretical importance. More precisely, it can be used to define binary products and coproducts, but the details exceed the scope of this dissertation.

Hence, the *type formation* rule is simple: we already have unit types and we already know how to make a coproduct. Therefore, the type $\mathbf{2}$ is readily made for us from previous constructions.

Being a coproduct type, its constructors are the injections, inl and inr, which in this case, apply to the sole element of the unit type: $\mathrm{inl}(*) : \mathbf{2}$ and $\mathrm{inr}(*) : \mathbf{2}$. In order to differentiate the components, we will denote by $0_{\mathbf{2}}$ the left element and by $1_{\mathbf{2}}$ the right element.

Now for the eliminator, i.e. to show how inhabitants of this type are used, we need a function $f : \mathbf{2} \longrightarrow C$, for some type $C : \mathcal{U}$. From the uniqueness principle, we know that there are only two elements of type $\mathbf{2}$, so in order to define such a function, it is sufficient to choose two elements, say $c_0, c_1 : C$. Then we define by a simple exhaustive case analysis:

$$f(0_{\mathbf{2}}) :\equiv c_0, \quad f(1_{\mathbf{2}}) :\equiv c_1.$$

We continue by noting that the recursor for this type corresponds to a simple branching (`if-then-else`) statement in programming or in other words, a two-case `switch` statement. That is, we have:

$$\mathrm{rec}_{\mathbf{2}} : \prod_{C:\mathcal{U}} C \longrightarrow C \longrightarrow \mathbf{2} \longrightarrow C,$$

which is defined by the equations (again, exhaustive case analysis):

$$\begin{cases} \mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) & :\equiv c_0 \\ \mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) & :\equiv c_1 \end{cases}.$$

In words, the "`switched`" statement has a dependent function type $\prod_{C:\mathcal{U}} C \to C \to \mathbf{2}$, which is basically a correspondence (mix) between the elements c_0, c_1 and if this correspondence ends up in $0_{\mathbf{2}} : \mathbf{2}$, then we return c_0, otherwise we return c_1.

In Pie syntax, since we already have the coproduct type at our disposal, with the `Either` type constructor, as well as the unit type, which is `Trivial`, the Boolean type is `(Either Trivial Trivial)`. If we take, say, the left component to mean false and the right, to mean true, then `false` is `(left sole)` and `true` is `(right sole)`.

In order to implement this in Pie, let us first make some shorthand notations in accordance to what we discussed above:

```
#lang pie

(claim Bool U)              ; claim the Bool type
(define Bool
  (Either Trivial Trivial)) ; Bool (2) is 1 (Trivial) + 1 (Trivial)

;; the truth values
(claim true Bool)
(define true (left sole))
(claim false Bool)
(define false (right sole))
```

Now we can define the recursor for this type, which is actually an `if` statement:

```
;; ... (the snippet above) ...

(claim if
  (Pi ((A U))
    (-> Bool A A A)))
(define if
  (lambda (A expr f t)
    (ind-Either expr
      (lambda (_) A)
      (lambda (_) f)
      (lambda (_) t))))
```

In order to explain this definition, first recall the general form of `ind-Either` from equation (3.2). What we're testing is `expr`, which has type `Bool` (i.e. `(Either Trivial Trivial)`),

then the motive chooses something of type `A` every time and the last two `lambda`s are the `base-left` and the `base-right` cases. That is, if what we're testing is some `(left x)`, it returns `f` and if it is some `(right y)`, it returns `t`. But recall that since `Bool` the sum type of false and true, the `left` and `right` statements are actually the false, respectively the true branches of an `if` statement.

We can test this readily like so:

```
;; ... (the snippet above) ...
(if Nat true zero 1)            ; => (the Nat 0)
(if Nat false zero 1)           ; => (the Nat 1)

;; unfolded syntax
(if Nat (left sole) 5 2)        ; => (the Nat 5)
(if Nat (right sole) 5 2)       ; => (the Nat 2)
```

APPENDIX: BASIC RACKET SYNTAX

Historically speaking, Racket is based on Scheme, being formerly called PLT Scheme. As such, it expands on the standards R5RS (1998) and R6RS (2009) of Scheme and diverge more significantly from R7RS (2013). However, we will not be concerned about the differences between the languages and we will assume that all Racket syntax that we introduce is valid Scheme syntax and vice versa, unless explicitly mentioned otherwise.

Both languages emerged from a common ancestor, Lisp and as such, they use a *list syntax*. Furthermore, they use a prefix notation for the functions, predicates and mathematical operations, the so-called Polish (Łukasiewicz) notation.

The only delimiters Scheme uses are parentheses (and double quotations for strings) whereas Racket strongly suggests the use of square brackets inside parentheses for delimiting more important statements[7]

Loosely speaking, both Scheme and Racket are untyped, but they do recognize three basic types: *lists, functions* and *symbols*, the latter encompassing variables and "everything else".

A specific feature is the so-called *quoting mechanism*, denoted by a single quote (`'`) or a backtick (`` ` ``) which turns anything into a symbol and the reverse, the *unquoting mechanism*, denoted by a comma (`,`) which enforces evaluation, as in the case of a call by name versus call by value approach. For example, `'(+ 1 2)` is interpreted as the symbol (string) `(+ 1 2)`, whereas `,(+ 1 2)` means `3`.

For simple expressions, quoting can be done either with a single quote or with a backtick. However, in more complex expressions, a difference appears. The backtick is actually called *quasiquote* and it can be used in expressions containing unquote. So basically we have:

```
'(+ 1 2)                ;; => (+ 1 2)

(define x 3)
```

[7] Some Scheme implementations allow the use of square brackets as well, but do not enforce it and furthermore, there are interpreters that reject this syntax.

```
`(+ 2 ,x)                ;; => 5      (used quasiquote)
'(+ 2 ,x)                ;; error    (used quote)
```

Moreover, there is also the *splice unquote* syntax, which is used to unquote lists. Instead of returning the whole list, as a regular unquote would do, splice unquote, denoted by `,@`, actually inserts the elements of the list:

```
(define x '(1 2 3))
`(+ 4 ,@x)               ;; => 10
```

Notice also the use of the quote in the first line, because we are just defining a (literal) list to store in `x`, whereas in the second line, we used the quasiquote, since the expression contains an unquote (the spliced one).

Another aspect of syntax is that a semicolon is used for comments and the convention is to use a single semicolon for an inline comment and two or more semicolons for comments spanning multiple lines. The output of a program is commonly written as `;; => output` inside the source code or documentation.

Remark 3.1: A short word on implementation: all the examples that we will be showcasing, as well as the included source code is tested on a Manjaro Linux operating system using the Emacs 26 editor, the included Scheme mode and the third party Racket mode.

After writing the source code, `C-c` loads the file into the respective REPL.

Some simple examples follow.

```
(+ (* 5 3) 1)        ;; => 16

(define (mod2 x)
  (lambda (x) (rem x 2)))
(mod2 5)             ;; => 1

(define (mod3 x)
  "Write the remainder of x when divided by 3"
  (cond                                          ; multiple branching
     ((equal (rem x 3) 1) write "it's 1")        ; if (x % 3 == 1)
     ((equal (rem x 3) 2) write "it's 2")        ; if (x % 3 == 2)
     (#t write "it's 0")))                       ; default (true) case

(define (add-or-quote x)
  "Add 3 if x is even or write 'hello' else"
  (cond
    ((equal (mod2 x) 0) (lambda (x) ,(+ 1 2)))
    (#t (lambda (x) 'hello))))

(set sum '(+ 1 2 3))         ; defines the variable "sum"
(set x (* ,sum 2))           ; defines x to be 12
```

Hopefully, the rest of the syntax can be understood directly from the examples that will follow. For further investigation, we recommend the official Racket documentation [2] and the book [6]. As the need requires, we will also further explain the syntax.

INDEX

BIBLIOGRAPHY

[1] The PLT Group. https://racket-lang.org/people.html, 2020. Accessed: March 2020.

[2] Racket. https://racket-lang.org/, 2020. Accessed: March 2020.

[3] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[4] David Thrane Christiansen.

[5] The Racket Community. Contracts documentation. https://docs.racket-lang.org/guide/contracts.html. Accessed March 2020.

[6] Matthias Felleisen and Bruce Findler. *How to Design Programs*. MIT Press, 2014.

[7] Daniel Friedman and David Christiansen. *The Little Typer*. MIT Press, 2018.

[8] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1990.

[9] Robert Goldblatt. *Topoi: The Categorical Analysis of Logic*. Dover, 2006.

[10] The HoTT Group. Homotopy Type Theory. https://homotopytypetheory.org/. Accessed March 2020.

[11] Arend Heyting. *Intuitionism, an Introduction*. Dover, 1966.

[12] Andrei Kolmogorov. Zur Deutung der intuitionistichen Logik. *Mathematische Zeitschrift*, 1932.

[13] Saunders MacLane and Ieke Moerdijk. *Sheaves in Geometry and Logic*. Springer, 1994.

[14] Per Martin-Löf. Intuitionistic Type Theory. Lectures in Padua, 1980. Notes by Giovanni Sambin.

[15] Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack, editors, *Types for Proofs and Programs*, pages 197–216, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[16] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[17] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Commun. ACM*, 3(4):184–195, April 1960.

[18] John McCarthy. A Basis for a Mathematical Theory of Computation. *Western Joint Computer Conference*, 1961.

[19] John McCarthy. Towards a Mathematical Science of Computation. *IFIP-62*, 1962.

[20] Colin McLarty. The uses and abuses of the history of topos theory. *The British Journal for the Philosophy of Science*, 41(3):351–375, 1990.

[21] Bengt Nordström and Kent Petersson. *Programming in Martin-Löf Type Theory*. Oxford University Press, 1990. Freely available at http://www.cse.chalmers.se/research/group/logic/book/.

[22] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[23] Prabhakar Ragde. Proust: A Nano Proof Assistant. In Johan Jeuring and Jay McCarthy, editors, *Trends in Functional Programming in Education*, pages 63–75. arXiv/cs.PL, 2016.

[24] Prabhakar Ragde. Personal conversation, 2020.

[25] Egbert Rijke. Introduction to Homotopy Type Theory. http://www.andrew.cmu.edu/user/erijke/hott/. Course notes, accessed March 2020.

[26] Morten Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier Science, 2006.

[27] Gerald Sussman et al. Revised7 Report on the Algorithmic Language Scheme. Technical report, Scheme Working Group, 7 2013.

[28] The CoqHoTT Team at INRIA. The Coq HoTT Project. http://coqhott.gforge.inria.fr/. Accessed April 2020.

[29] Various. The HoTT-Agda Project. https://github.com/HoTT/HoTT-Agda. Accessed April 2020.