Boro Sitnikovski

λ Π

GENTLE INTRODUCTION TO DEPENDENT TYPES WITH IDRIS

Gentle Introduction to Dependent Types with Idris

Boro Sitnikovski

This book is for sale at http://leanpub.com/gidti

This version was published on 2022-01-03

dedicated to my wife Dijana, and our kids

Contents

Preface and acknowledgments **1**

Introduction **3**

1. Formal systems **4**

1.1. MU puzzle example 6

2. Classical mathematical logic **8**

2.1. Hierarchy of mathematical logic and definitions 8

2.1.1. Propositional logic 8

2.1.2. First-order logic 11

2.1.3. Higher-order logic 12

2.2. Set theory abstractions 14

2.3. Substitution and mathematical proofs 18

2.3.1. Proofs by truth tables 20

2.3.2. Three-column proofs 21

2.3.3. Formal proofs 22

2.3.4. Mathematical induction 24

3. Type theory **26**

3.1. Lambda calculus 29

3.1.1. Term reduction 30

3.2. Lambda calculus with types 32

3.3. Dependent types 35

3.4. Intuitionistic theory of types 37

3.4.1. Intuitionistic logic 40

4. Programming in Idris **42**

4.1. Basic syntax and definitions . . . 42
4.1.1. Defining functions . . . 43
4.1.2. Defining and inferring types . . . 44
4.1.3. Anonymous lambda functions . . . 49
4.1.4. Recursive functions . . . 51
4.1.5. Recursive data types . . . 54
4.1.6. Total and partial functions . . . 56
4.1.7. Higher-order functions . . . 57
4.1.8. Dependent types . . . 59
4.1.9. Implicit parameters . . . 61
4.1.10. Pattern matching expressions . . . 63
4.1.11. Interfaces and implementations . . . 64
4.2. Curry-Howard isomorphism . . . 65
4.3. Quantitative Type Theory . . . 67

5. Proving in Idris . . . 68
5.1. Weekdays . . . 69
5.1.1. First proof (auto-inference) . . . 69
5.1.2. Second proof (rewrite) . . . 71
5.1.3. Third proof (impossible) . . . 73
5.2. Natural numbers . . . 75
5.2.1. First proof (auto-inference and existence) . . . 75
5.2.2. Second proof (introduction of a new given) . . . 76
5.2.3. Third proof (induction) . . . 78
5.2.4. Ordering . . . 80
5.2.5. Safe division . . . 81
5.2.6. Maximum of two numbers . . . 83
5.2.7. List of even naturals . . . 86
5.2.8. Partial orders . . . 88
5.3. Computations as types . . . 90
5.3.1. Same elements in a list (vector) . . . 90
5.3.2. Evenness of numbers . . . 93
5.4. Trees . . . 94
5.4.1. Depth . . . 95
5.4.2. Map and size . . . 97
5.4.3. Length of mapped trees . . . 98

Conclusion **100**

Further reading **101**

Appendices **102**
Appendix A: Writing a simple type checker in Haskell 102
Evaluator 102
Type checker 104
Environments 105
Appendix B: Theorem provers 108
Metamath 108
Simple Theorem Prover 111
Appendix C: IO, Codegen targets, compilation, and FFI 113
IO 113
Codegen 115
Compilation 116
Foreign Function Interface 119
Appendix D: Implementing a formal system 122

About the author **124**

Preface and acknowledgments

This book aims to be accessible to novices that have no prior experience beyond high school mathematics. Thus, this book is designed to be self-contained. No programming experience is assumed, however having some kind of programming experience with the functional programming paradigm could make things easier to grasp in the beginning. After you finish reading the book I recommend that you check the "Further reading" chapter in case you are interested in diving deeper into some of the topics discussed.

I was always curious to understand how things work. As a result, I became very interested in mathematics while I was in high school. One of the reasons for writing this book is that I could not find a book that explained how things work, so I had to do a lot of research on the internet through white-papers, forums, and example code in order to come up with a complete picture of what dependent types are and what they are good for.

I will consider this book successful if it provides you with some additional knowledge. I tried to write this book so that the definitions and examples provided in it show how different pieces of the puzzle are connected to each other.

Feel free to contact me at buritomath@gmail.com for any questions you might have, and I will do my best to answer. You can also access my blog at bor0.wordpress.com to check out some of my latest work.

Book reviewers:

1. Nathan Bloomfield did a Ph.D. in algebra at the University of Arkansas and taught mathematics before joining Automattic, Inc. as a programmer. He enjoys witnessing the Intuitionist renaissance and likes how the boring parts of abstract algebra transform into really interesting computer science. Nathan usually feels a little out of his depth and prefers it that way.
2. Vlad Riscutia is a software engineer at Microsoft working on Office. He is interested in programming languages and type systems, and how these can best be leveraged to write correct code.

3. Marin Nikolovski is working at Massive Entertainment | A Ubisoft Studio as a Senior Web Developer on UPlay PC. He has over 10 years of experience with designing, developing and testing software across a variety of platforms. He takes pride in coding to consistently high standards and constantly tries to keep up the pace with the latest developments in the IT industry.
4. Neil Mitchell is a Haskell programmer with a Ph.D. in Computer Science from York University, where he worked on making functional programs shorter, faster and safer. Neil is the author of popular Haskell tools such as Hoogle, HLint, and Ghcid - all designed to help developers write good code quickly. More recently Neil has been researching build systems with publications at ICFP and the Haskell Symposium, and a practical system based on those ideas named Shake.

Thanks to:

1. Irena Jazeva for the book cover design
2. The Haskell community (#haskell@freenode)
3. The Idris community (#idris@freenode)
4. The Coq community (#coq@freenode)

Thanks to my family, coworkers, and friends for all the support they give to me.

Finally, thank you for purchasing this book! I hope that you will learn new techniques in reading this book and that it will spark up some more interest in logic, dependent types, and type theory.

Introduction

Writing correct code in software engineering is a complex and expensive task, and too often our written code produces inaccurate or unexpected results. There are several ways to deal with this problem. In practice, the most common approach is to write *tests*, which means that we are writing more code to test our original code. However, these tests can only ever detect problems in specific cases. As Edsger Dijkstra noted, "testing shows the presence, not the absence of bugs". A less common approach is to find a *proof of correctness* for our code. A software proof of correctness is a logical proof that the software is functioning according to given specifications. With valid proofs, we can cover all possible cases and be more confident that the code does exactly what it is intended to do.

Idris is a general-purpose functional[1] programming language that supports dependent types. The features of Idris are influenced by Haskell, another general-purpose functional programming language. Thus, Idris shares many features with Haskell, especially in the part of syntax and types, where Idris has a more advanced type system. There are several other programming languages that support dependent types[2], however, I chose Idris for its readable syntax.

The first version of Idris was released in 2009. It is developed by The Idris Community and led by Dr. Edwin Brady. Seen as a programming language, it is a functional programming language implemented with dependent types. Seen as a logical system, it implements intuitionistic type theory, which we will cover later. Finally, we will show how these two views relate to each other in section 4.2.

Idris allows us to express mathematical statements. By mechanically examining these statements, it helps us find a formal proof of a program's formal specification.

To fully understand how proofs in Idris work we will start with the foundations by defining: formal systems, classical mathematical logic, lambda calculus, intuitionistic logic, and type theory (which is a more "up-to-date" version of classical mathematical logic).

[1] The core concept of functional programming languages is a mathematical function.

[2] Several other languages with dependent types support are Coq, Agda, Lean.

1. Formal systems

Before we can construct proofs of correctness for software we need to understand what a proof is and what it means for a proof to be valid. This is the role of *formal systems*. The purpose of formal systems is to let us reason about reasoning – to manipulate logical proofs in terms of their *form*, rather than their *content*. This level of abstraction makes formal systems powerful tools.

Definition 1

A **formal system** is a model of abstract reasoning. A formal system consists of:

1. A **formal language** that contains:
    1. A finite set of *symbols*, which can be combined into finite strings called *formulas*
    2. A *grammar*, which is a set of rules that tells us which formulas are "well-formed"
2. A set of **axioms**, that is, formulas we accept as "valid" without justification
3. A set of **inference rules** that tell us how we can derive new valid formulas from old ones

Inside a given formal system the grammar determines which formulas are *syntactically* sensible, while the inference rules govern which formulas are *semantically* sensible. The difference between these two is important. For example, thinking of the English language as a (very complicated!) formal system, the sentence "Colorless green ideas sleep furiously" is syntactically valid (since different parts of speech are used in the right places), but is semantically nonsense.

After a formal system is defined, other formal systems can extend it. For example, set theory is based on first-order logic, which is based on propositional logic which represents a formal system. We'll discuss this theory briefly in the next chapter.

Definition 2

For a given formal system, the system is **incomplete** if there are statements that are true but which cannot be proved to be true inside that system. Conversely, the system is **complete** if all true statements can be proved.

The statement “This statement is not provable” can either be true or false. In the case it is true, then it is not provable. Alternatively, in the case it is false then it is provable, but we’re trying to prove something false. Thus the system is incomplete because some truths are unprovable.

Definition 3

For a given formal system, the system is **inconsistent** if there is a theorem in that system that is contradictory. Conversely, the system is **consistent** if there are no contradictory theorems.

A simple example is the statement “This statement is false”. This statement is true if and only if[3] it is false, and therefore it is neither true nor false.

In general, we often put our focus on which parts of mathematics can be formalized in concrete formal systems, rather than trying to find a theory in which all of mathematics can be developed. The reason for that is Gödel’s incompleteness theorem. This theorem states that there doesn’t exist[4] a formal system that is both complete and consistent. As a result, it is better to reason about a formal system outside of the system (at the metalanguage level), since the object level (rules within the system) can be limiting. As the famous saying goes to “think outside of the box”, and similarly to how we sometimes do meta-thinking to improve ourselves.

In conclusion, formal systems are our attempt to abstract models, whenever we reverse engineer nature in an attempt to understand it better. They may be imperfect but are nevertheless useful tools for reasoning.

[3] The word iff is an abbreviation for “If and only if” and means that two statements are logically equivalent.

[4] Note that this theorem only holds for systems that allow expressing arithmetic of natural numbers (e.g. Peano, set theory, but first-order logic also has some paradoxes if we allow self-referential statements). This is so because the incompleteness theorem relies on the Gödel numbering concept, which allows a formal system to reason about itself by using symbols in the system to map expressions in that same system. For example, 0 is mapped to 1, S is 2, = is 3, so $0 = S0 \iff (1, 3, 2, 1)$. Using this we can express statements about the system within the system - self-referential statements. We will look into these systems in the next chapter.

1.1. MU puzzle example

The MU puzzle is a formal system that we'll have a look at as an example.

Definition 4

We're given a starting string `MI`, combined with a few inference rules, or transformation rules:

No.	Rule	Description	Example
1	`xI` → `xIU`	Append `U` at a string ending in `I`	`MI` to `MIU`
2	`Mx` → Mxx	Double the string after `M`	`MIU` to `MIUIU`
3	`xIIIy` → `xUy`	Replace `III` inside a string with `U`	`MUIIIU` to `MUUU`
4	`xUUy` → xy	Remove `UU` from inside a string	`MUUU` to `MU`

In the inference rules the symbols `M`, `I`, and `U` are part of the system, while `x` is a variable that stands for any symbol(s). For example, `MI` matches rule 2 for `x = I`, and it can also match rule 1 for `x = M`. Another example is `MII` that matches rule 2 for `x = II` and rule 1 for `x = MI`.

We will show (or prove) how we can get from `MI` to `MIIU` using the inference rules:

1. `MI` (axiom)
2. `MII` (rule 2, `x = I`)
3. `MIIII` (rule 2, `x = II`)
4. `MIIIIIIII` (rule 2, `x = IIII`)
5. `MUIIIII` (rule 3, `x = M`, `y = IIIII`)
6. `MUUII` (rule 3, `x = MU`, `y = II`)
7. `MII` (rule 4, `x = M`, `y = II`)
8. `MIIU` (rule 1, `x = MI`)

We can represent the formal description of this system as follows:

1. Formal language

   1. Set of symbols is $\{M, I, U\}$
   2. A string is well-formed if the first letter is `M` and there are no other `M` letters. Examples: `M`, `MIUIU`, `MUUUIII`
2. `MI` is the starting string, i.e. axiom
3. The rules of inference are defined in Definition 4

Can we get from `MI` to `MU` with this system?

To answer this, we will use an invariant[5] with mathematical induction to prove our claim.

Note that, to be able to apply rule 3, we need to have the number of subsequent `I`'s to be divisible by 3. Let's have our invariant say that "There is no sequence of `I`'s in the string that with length divisible by 3":

1. For the starting axiom, we have one `I`. Invariant OK.
2. Applying rule 2 will be doubling the number of `I`'s, so we can have: `I`, `II`, `IIII`, `IIIIIII` (in particular, 2^n `I`'s). Invariant OK.
3. Applying rule 3 will be reducing the number of `I`'s by 3. But note that $2^n - 3$ is still not divisible by 3[6]. Invariant OK.

We've shown that with the starting axiom `MI` it is not possible to get to `MU` because no sequence of steps can turn a string with one `I` into a string with no `I`s. But if we look carefully, we've used a different formal system to reason about `MU` (i.e. divisibility by 3, which is not part of the MU system). This is because the puzzle cannot be solved in its own system. Otherwise, an algorithm would keep trying different inference rules of `MU` indefinitely (not knowing that `MU` is impossible).

Every useful formal system has this limitation. As we've seen, Gödel's theorem shows that there's no formal system that can contain all possible truths, because it cannot prove some truths about its own structure. Thus, having experience with different formal systems and combining them as needed can be useful.

[5] An invariant is a property that holds whenever we apply any of the inference rules.

[6] After having introduced ourselves to proofs, you will be given an exercise to prove this fact.

2. Classical mathematical logic

All engineering disciplines involve some usage of logic. The foundations of Idris, as we will see later, are based on a system that implements (or encodes) classical mathematical logic so that we can easily “map” this logic and its inference rules to computer programs.

2.1. Hierarchy of mathematical logic and definitions

At its core, mathematical logic deals with mathematical concepts expressed using formal logical systems. In this section, we’ll take a look at the hierarchy of these logical systems. The reason why we have different levels of hierarchies is that at each level we have more power in expressiveness. Further, these logical systems are what will allow us to produce proofs.

2.1.1. Propositional logic

Definition 1

The propositional branch of logic is concerned with the study of **propositions**, which are statements that are either $\top$ (true) or $\bot$ (false). Variables can be used to represent propositions. Propositions are formed by other propositions with the use of logical connectives. The most basic logical connectives are $\land$ (and), $\lor$ (or), $\neg$ (negation), and $\to$ (implication).

For example, we can say `a = Salad is organic`, and thus the variable `a` represents a true statement. Another statement is `a = Rock is organic`, and thus `a` is a false statement. The statement `a = Hi there!` is neither a true nor a false statement, and thus is not a proposition.

The "and" connective means that both a and b have to be true in order for $a \wedge b$ to be true. For example, the statement `I like milk and sugar` is true as a whole iff both `I like milk` and `I like sugar` are true.

a	**b**	$a \wedge b$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

The "or" connective means that either of a or b has to be true in order for $a \vee b$ to be true. It will also be true if both a and b are true. This is known as inclusive or. For example, the statement `I like milk or sugar` is true as a whole if at least one of `I like milk` or `I like sugar` is true.

This definition of "or" might be a bit counter-intuitive to the way we use it in day to day speaking. When we say `I like milk or sugar` we normally mean one of them but not both. This is known as exclusive or, however, for the purposes of this book we will be using inclusive or.

a	**b**	$a \vee b$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

The negation connective simply swaps the truthiness of a proposition. The easiest way to negate any statement is to just prepend `It is not the case that ...` to it. For example, the negation of `I like milk` is `It is not the case that I like milk`, or simply `I don't like milk`.

a	$\neg a$
$\top$	$\bot$
$\bot$	$\top$

The implication connective allows us to express conditional statements, and its interpretation is subtle. We say that $a \to b$ is true if anytime a is true, it is necessarily also the case that b is true. Another way to think about implication is in terms of *promises*; $a \to b$ represents a promise that if a happens, then b also happens. In this interpretation, the truth value of $a \to b$ is whether or not the promise is kept, and we say that a promise is kept unless it has been broken.

For example, if we choose `a = Today is your birthday` and `b = I brought you a cake`, then $a \to b$ represents the promise `If today is your birthday, then I brought you a cake`. Then there are four different ways that today can play out:

1. Today is your birthday, and I brought you a cake. The promise is kept, so the implication is true
2. Today is your birthday, but I did not bring you a cake. The promise is not kept, so the implication is false
3. Today is not your birthday, and I brought you a cake. Is the promise kept? Better question - has the promise been broken? The condition the promise is based on - that today is your birthday - is not satisfied, so we say that the promise is not broken. The implication is true
4. Today is not your birthday, and I did not bring you a cake. Again, the condition of the promise is not satisfied, so the promise is not broken. The implication is true

In the last two cases, where the condition of the promise is not satisfied, we sometimes say that the implication is *vacuously true*.

This definition of implication might be a bit counter-intuitive to the way we use it in day to day speaking. When we say `If it rains, then the ground is wet` we usually mean both that `If the ground is wet, then it rains` and `If it rains, then the ground is wet`. This is known as biconditional and is denoted as $a \leftrightarrow b$, or simply a iff b.

a	**b**	$a \to b$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$

As stated in Definition 1, propositions can also be defined (or combined) in terms of other propositions. For example, we can choose a to be `I like milk` and b to be `I like sugar`. So $a \land b$ means that `I like both milk and sugar`. If we let c be `I am` `cool` then with $a \land b \to c$ we say: `If I like milk and sugar, then I am cool.` Note how we took a proposition $a \land b$ and modified it with another connective to form a new proposition.

Exercise 1

Come up with a few propositions and combine them using:

1. The "and" connective
2. The "or" connective
3. Negation connective
4. Implication connective

Try to come up with a sensible statement in English for each derived proposition.

2.1.2. First-order logic

Definition 2

The first-order logical system extends propositional logic by additionally covering **predicates** and **quantifiers**. A predicate $P(x)$ takes an input x, and produces either true or false as an output. There are two quantifiers introduced: $\forall$ (universal quantifier) and $\exists$ (existential quantifier).

One example of a predicate is `P(x) = x is organic`, with $P(\mathit{Salad}) = \top$, but $P(\mathit{Rock}) = \bot$.

In the following example the universal quantifier says that the predicate will hold for **all** possible choices of x: $\forall x, P(x)$. Alternatively, the existential quantifier says that the predicate will hold for **at least one** choice of x: $\exists x, P(x)$.

Another example of combining a predicate with the universal quantifier is `P(x) = x is a mammal`, then $\forall x, P(x)$ is true, for all x ranging over the set of humans. We can choose `P(x) = x understands Dependent Types` with $\exists x, P(x)$ to say that there is at least one person that understands Dependent Types.

The negation of the quantifiers is defined as follows:

1. Negation of universal quantifier: $\neg(\forall x, P(x)) \leftrightarrow \exists x, \neg P(x)$
2. Negation of existential quantifier: $\neg(\exists x, P(x)) \leftrightarrow \forall x, \neg P(x)$

As an example, for `P(x) = x understands Dependent Types` the negation of $\exists x, P(x)$ is $\forall x, \neg P(x)$. That is, the negation of `there is at least one person that understands Dependent Types` is `for all persons x, x does not understand Dependent Types`, or simply put `nobody understands Dependent Types`.

Exercise 2

Think of a real-world predicate and express its truthiness using the $\forall$ and $\exists$ symbols. Afterward, negate both the universal and existential quantifier.

2.1.3. Higher-order logic

In first-order logic, predicates act like functions that take an input value and produce a proposition. A predicate can't be true or false until a specific value is substituted for the variables, and the quantifiers $\forall$ and $\exists$ "close" over a predicate to give a statement which can be either true or false.

Likewise, we can define a "meta-predicate" that acts as a function on predicates. For example, let $\Gamma(P)$ be the statement `there exists a person x such that P(x) is true`. Note that it doesn't make sense to ask if $\Gamma(P)$ is true or false until we plug in a specific *predicate* P. But we can quantify over P, and construct a statement like $\forall P, \Gamma(P)$. In English, this statement translates to `For any given property P, there exists a person satisfying that property`.

Meta-predicates like Γ are called *second-order* because they range over first-order predicates. And there's no reason to stop there; we could define third-order predicates that range over second-order predicates, and fourth-order predicates that range over third-order predicates, and so on.

Definition 3

The higher-order logical system [second-order logic, third-order-logic, ..., higher-order (nth-order) logic] extends the quantifiers that range over individuals[7] to range over predicates.

For example, the second-order logic quantifies over sets. Third-order logic quantifies over sets of sets, and so on.

Moving up the hierarchy of logical systems brings power, at a price. Propositional (zeroth-order) logic is completely decidable[8]. Predicate (first-order) logic is no longer decidable, and by Gödel's incompleteness theorem we have to choose between completeness and consistency, but at least there is still an algorithm that can determine whether a proof is valid or not. For second-order and higher logic we lose even this - we have to choose between completeness, consistency, and a proof detection algorithm.

The good news is that in practice, second-order predicates are used in a very limited capacity, and third- and higher-order predicates are never needed. One important example of a second-order predicate appears in the Peano axioms of the natural numbers.

Definition 4

Peano's axioms is a system of axioms that describes the natural numbers. It consists of 9 axioms, but we will name only a few:

1. 0 (zero) is a natural number
2. For every number x, we have that $S(x)$ is a natural number, namely the successor function
3. For every number x, we have that $x = x$, namely that equality is reflexive

[7] Since unrestricted quantification leads to inconsistency, higher-order logic is an attempt to avoid this. We will look into Russell's paradox later as an example.

[8] This means that there is a decidability algorithm - an algorithm that will always return a correct value (e.g. true or false), instead of looping infinitely or producing a wrong answer.

Definition 5

The ninth axiom in Peano's axioms is the induction axiom. It states the following: if P is a predicate where $P(0)$ is true, and for every natural number n if $P(n)$ is true then we can prove that $P(n+1)$, then $P(n)$ is true for all natural numbers.

Peano's axioms are expressed using a combination of first-order and second-order logic. This concept consists of a set of axioms for the natural numbers, and all of them are statements in first-order logic. An exception of this is the induction axiom, which is in second-order since it quantifies over predicates. The base axioms can be augmented with arithmetical operations of addition, multiplication and the order relation, which can also be defined using first-order axioms.

2.2. Set theory abstractions

Definition 6

Set theory is a type of a formal system, which is the most common **foundation of mathematics**. It is a branch of mathematical logic that works with **sets**, which are collections of objects.

Like in programming, building abstractions in mathematics is of equal importance. However, the best way to understand something is to get to the bottom of it. We'll start by working from the lowest level to the top. We will start with the most basic object (the unordered collection) and work our way up to defining functions. Functions are an important core concept of Idris, however, as we will see in the theory that Idris relies on, functions are used as a primitive notion (an axiom) instead of being built on top of something else.

Definition 7

A set is an **unordered** collection of objects. The objects can be anything.

Finite sets can be denoted by *roster notation*; we write out a list of objects in the set, separated by commas, and enclose them using curly braces. For example, one set of fruits is $\{\text{apple}, \text{banana}\}$. Since it is an unordered collection we have that $\{\text{apple}, \text{banana}\} = \{\text{banana}, \text{apple}\}$.

Definition 8

Set membership states that a given object belongs to a set. It is denoted using the $\in$ operator.

For example, $\text{apple} \in \{\text{apple}, \text{banana}\}$ says that apple is in that set.

Roster notation is inconvenient for large sets, and not possible for infinite sets. Another way to define a set is with *set-builder notation*. With this notation, we specify a set by giving a predicate that all of its members satisfy. A typical set in set-builder notation has the form $\{x \mid P(x)\}$, where P is a predicate. If a is a specific object, then $a \in \{x \mid P(x)\}$ precisely when $P(a)$ is true.

Definition 9

An n-tuple is an **ordered collection** of n objects. As with sets, the objects can be anything. Tuples are usually denoted by comma separating the list of objects and enclosing them using parentheses.

For example, we can use the set $\{\{1, \{a_1\}\}, \{2, \{a_2\}\}, \ldots, \{n, \{a_n\}\}\}$ to represent the ordered collection $(a_1, a_2, ..., a_n)$. This will now allow us to extract the k-th element of the tuple, by picking x such that $\{k, \{x\}\} \in A$. Having done that, now we have that $(a, b) = (c, d) \equiv a = c \land b = d$, that is, two tuples are equal iff their first and second elements respectively are equal. This is what makes them ordered.

One valid tuple is $(1 \text{ pm}, 2 \text{ pm}, 3 \text{ pm})$ which represents 3 hours of a day sequentially.

Definition 10

An n-ary relation is just a set of n-tuples.

For example, the `is bigger than` relation represents a 2-tuple (pair), for the following set: $\{(\text{cat}, \text{mouse}), (\text{mouse}, \text{cheese}), (\text{cat}, \text{cheese})\}$.

Definition 11

A is a subset of B if all elements of A are found in B (but not necessarily vice-versa). We denote it as such: $A \subseteq B$.

For example, the expressions $\{1, 2\} \subseteq \{1, 2, 3\}$ and $\{1, 2, 3\} \subseteq \{1, 2, 3\}$ are both true. But this expression is not true: $\{1, 2, 3\} \subseteq \{1, 2\}$.

Definition 12

A Cartesian product is defined as the set $\{(a, b) \mid a \in A \land b \in B\}$. It is denoted as $A \times B$.

For example if $A = \{a, b\}$ and $B = \{1, 2, 3\}$ then the combinations are: $A \times B = \{(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3)\}$.

Definition 13

Functions are defined in terms of relations[9]. A binary (2-tuple) set F represents a mapping[10] from some set A to some set B, where F is a subset of the Cartesian product of A and B. That is, a function f from A to B is denoted $f : A \to B$ and is a subset of F, i.e. $f \subseteq F$. There is one more constraint that functions have, namely, that they cannot produce 2 or more different values for a single input.

For example, the function $f(x) = x + 1$ is a function that, given a number, returns it increased by one. We have that $f(1) = 2$, $f(2) = 3$, etc. Another way to represent this function is using the 2-tuple set: $f = \{(1, 2), (2, 3), (3, 4), \ldots\}$.

One simple way to think of functions is in the form of tables. For a function $f(x)$ accepting a single parameter x, we have a two-column table where the first column

[9] It is worth noting that in set theory, P would be a subset of a relation, i.e. $P \subseteq A \times \{T, F\}$, where A is a set of some inputs, for example `Salad` and `Rock`. When working with other systems we need to be careful, as this is not the case with first-order logic. In the case of first-order logic, we have $P(Salad) = \top$, $P(Rock) = \bot$, etc as atomic statements, not mathematical functions (i.e. they cannot be broken down into smaller statements). This is what makes first-order logic independent of set theory. In addition, functions have a nice characterization that is dual to the concepts of "one-to-one" (total) and "onto" (well-defined).

[10] In other words, a function is a subset of all combinations of ordered pairs whose first element is an element of A and the second element is an element of B.

is the input, and the second column is the output. For a function $f(x, y)$ accepting two parameters x and y we have a three-column table where the first and second columns represent the input, and the third column is the output. Thus, to display the function discussed above in the form of a table, it would look like this:

x	$f(x)$
1	2
2	3
…	…

Exercise 3

Think of a set of objects and express that some object belongs to that set.

Exercise 4

Think of a set of objects whose order matters and express it in terms of an ordered collection.

Exercise 5

Think of a relation (for example, a relation between two persons in a family tree) and express it using the notation described.

Exercise 6

Come up with two subset expressions, one that is true and another one that is false.

Exercise 7

Think of two sets and combine them using the definition of the Cartesian product. Afterward, think of two subset expressions, one that is true and another one that is false.

Exercise 8

Think of a function and represent it using the table approach.

Exercise 9

Write down the corresponding input and output sets for the function you implemented in Exercise 8.

2.3. Substitution and mathematical proofs

Substitution lies at the heart of mathematics[11].

Definition 14

Substitution consists of systematically replacing occurrences of some symbol with a given value. It can be applied in different contexts involving formal objects containing symbols.

For example, let's assume that we have the following:

1. An inference rule that states: If $a = b$ and $b = c$, then $a = c$
2. Two axioms that state: $1 = 2$ and $2 = 3$

We can use the following "proof" to claim that $1 = 3$:

1. $1 = 2$ (axiom)
2. $2 = 3$ (axiom)
3. $1 = 2$ and $2 = 3$ (from 1 and 2 combined)
4. $1 = 3$, from 3 and the inference rule

[11] A similar statement can be made about programming, but we will cover an interesting case in Appendix C related to **pure** and **impure** functions.

We know that in general, $1 = 3$ does not make any sense. But, in the context of the givens above, this proof is valid.

Definition 15

A mathematical argument consists of a list of propositions. Mathematical arguments are used in order to demonstrate that a claim is true or false.

Definition 16

A proof is defined as an inferential **argument** for a list of given mathematical propositions. To prove a mathematical fact, we need to show that the conclusion (the goal that we want to prove) logically follows from the hypothesis (list of given propositions).

For example, to prove that a goal G follows from a set of given propositions $\{g_1, g_2, \ldots, g_n\}$, we need to show $(g_1 \land g_2 \land \ldots \land g_n) \to G$. Note the relation between the implication connective[12] (conditional statement) and proofs.

Exercise 10

With the given axioms of Peano, prove that $1 = S(0)$ and $2 = S(S(0))$ are natural numbers.

Exercise 11

Come up with several axioms and inference rules and do a proof similar to the example above.

[12] The turnstile symbol is similar to implication. It is denoted as $\Gamma \vdash A$, where Γ is a set of statements and A is a conclusion. It is $\top$ iff it is impossible for all statements in Γ to be $\top$, and A to be $\bot$. The reason why we have both implication and entailment is that the former is a well-formed formula (that is, the expression belongs to the object language), while the latter is not a well-formed formula, rather an expression in the metalanguage and works upon proofs (instead of objects).

2.3.1. Proofs by truth tables

Here's one claim: The proposition $A \land B \to B$ is true for **any** values of A and B.

How do you convince someone that this proposition is really true?

We can use one proof technique which is to construct a truth table. The way truth tables are constructed for a given statement is to break it down into atoms and then include every subset of the expression.

For example, to prove the statement $A \land B \to B$, we can approach as follows:

A	**B**	$A \land B$	$A \land B \to B$
$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$	$\top$

Definition 17

A mathematical argument is valid iff in the case where all of the propositions are true, the conclusion is also true.

Note that wherever $A \land B$ is true (the list of given propositions, or premises, or hypothesis) then so is $A \land B \to B$ (the conclusion), which means that this is a valid logical argument according to Definition 17.

At the lowest level (formal systems), proofs were just a transformation (substitution) from one expression to another. However, at a higher level (logic), the way proofs are done is closely related to the symbols defined in the system. For example, in logic, there's $\land$ so there are specific rules about introducing/eliminating it.

Exercise 12

Given the two propositions $A \lor B$ and $\neg B$, prove (or conclude) A by means of a truth table.

Hint: The statement to prove is $((A \lor B) \land \neg B) \to A$.

2.3.2. Three-column proofs

As we've defined before, an argument is a list of statements. There are several ways to do mathematical proofs. Another one of them is by using the so-called three-column proofs. For this technique, we construct a table with three columns: number of step, step (or expression derived), and reasoning (explanation of how we got to the particular step).

Definition 18

Modus ponens (method of affirming) and modus tollens (method of denying) are two inference rules in logic. Their definition is as follows:

1. Modus ponens states: If we are given $p \to q$ and p, then we can conclude q
2. Modus tollens states: If we are given $p \to q$ and $\neg q$, then we can conclude $\neg p$

For example, given $A \vee B$, $B \to C$, $\neg C$, prove A. We can approach the proof as follows:

No.	Step	Reasoning
1	$A \vee B$	Given
2	$B \to C$	Given
3	$\neg C$	Given
4	$(B \to C) \wedge \neg C$	2 and 3
5	$\neg B$	Modus tollens rule on 4, i.e. $(p \to q \wedge \neg q) \to \neg p$
6	$(A \vee B) \wedge \neg B$	1 and 5
7	A	6, where $p \wedge \neg p$ is a contradiction, i.e. invalid argument

Proofs with truth tables look a lot easier than column proofs. You just plug in truth values and simplify, where column proofs require planning ahead. Why would we bother with column proofs?

Proofs with truth tables only work for propositional (zeroth order) theorems - the table method is essentially the decidability algorithm for zeroth-order logic. That's why they are easy (if verbose) and always work, and why column proofs become necessary once we're using quantifiers.

Exercise 13

Prove $((A \lor B) \land \neg B) \to A$ using the three-column proof technique.

2.3.3. Formal proofs

We've seen how we can construct proofs with truth tables. However, if our statements involve the use of quantifiers, then doing proofs with truth tables is impossible. Three-column proofs, in contrast, contain many details. Ideally, the proof should be short, clear and concise about what we want to prove. Therefore, we will try to prove a statement by means of a formal proof.

To prove $A \land B \to B$, we start by assuming that $A \land B$ is true since otherwise, the statement is vacuously true by definition for implication. If $A \land B$ is true, then both A and B are true by definition of `and`, that is, we can conclude B.

Do not worry if the previous paragraph sounded too magical. There is not much magic involved. Usually, it comes down to using a few rules (or "tricks", if you will) for how we can use given information and achieve our goal. We will summarize these proof techniques next.

In order to **prove** a goal of form:

Goal form	Technique
$P \to Q$	Assume that P is true and prove Q
$\neg P$	Assume that P is true and arrive at a contradiction
$P_1 \land P_2 \land \ldots \land P_n$	Prove each one of $P_1, P_2, \ldots, P_n$ separately
$P_1 \lor P_2 \lor \ldots \lor P_n$	Prove that at least one of $P_1, P_2, \ldots, P_n$
$P \leftrightarrow Q$	Prove both $P \to Q$ and $Q \to P$
$\forall x, P(x)$	Assume that x is an arbitrary object and prove that $P(x)$
$\exists x, P(x)$	Find an x such that $P(x)$ is true
$\exists! x, P(x)$[13]	Prove $\exists x, P(x)$ (existence) and $\forall x \forall y, (P(x) \land P(y) \to x = y)$ (uniqueness) separately

[13] The notation ∃! stands for unique existential quantifier. It means that **only one** object fulfills the predicate, as opposed to ∃, which states that **at least one** object fulfills the predicate.

In order to **use** a given of form:

Given form	**Technique**
$P \to Q$	If P is also given, then conclude that Q (by modus ponens)
$\neg P$	If P can be proven true, then conclude a contradiction
$P_1 \wedge P_2 \wedge \ldots \wedge P_n$	Treat each one of $P_1, P_2, \ldots, P_n$ as a given
$P_1 \vee P_2 \vee \ldots \vee P_n$	Use proof by cases, where in each case you assume one of $P_1, P_2, \ldots, P_n$
$P \leftrightarrow Q$	Conclude both $P \to Q$ and $Q \to P$
$\forall x, P(x)$	For any x, conclude that $P(x)$
$\exists x, P(x)$	Introduce a new variable, say x_0 so that $P(x_0)$ is true
$\exists ! x, P(x)$	Introduce a new variable, say x_1 so that $P(x_1)$ is true. Can also use that $\forall x \forall y, (P(x) \wedge P(y) \to x = y)$

For example, we can use these techniques to do the following proofs:

1. $A \wedge B \to A \vee B$ - To prove this goal, we will assume $A \wedge B$ and use proof by cases:
    1. Proof for A: Since we're given $A \wedge B$, we are also given A. Thus, A
    2. Proof for B: Since we're given $A \wedge B$, we are also given B. Thus, B
    3. Thus, $A \vee B$
2. $A \wedge B \leftrightarrow B \wedge A$ - To prove this goal, we will prove both sides for the implications:
    1. Proof for $A \wedge B \to B \wedge A$: We can assume that $A \wedge B$, thus we have both A and B. To prove the goal of $B \wedge A$, we need to prove B and A separately, which we already have as given.
    2. Proof for $B \wedge A \to A \wedge B$: We can assume that $B \wedge A$, thus we have both B and A. To prove the goal of $A \wedge B$, we need to prove A and B separately, which we already have as given.
    3. Thus, $A \wedge B \leftrightarrow B \wedge A$
3. $\forall x, x = x$ - We know that for any number x, this number is equal to itself. Thus, $\forall x, x = x$.
4. $\exists x, x > 0$ - To prove this, we only need to find an x such that it is greater than 0. One valid example is 1. Thus, $\exists x, x > 0$.

Exercise 14

Prove $((A \vee B) \wedge \neg B) \to A$ by means of a formal proof.

Exercise 15

We've used the rules modus tollens and modus ponens without giving an actual proof for them. Try to prove by yourself that these two rules hold, both by constructing a truth table and a three-column proof:

1. Modus tollens: $((p \to q) \land \neg q) \to \neg p$
2. Modus ponens: $((p \to q) \land p) \to q$

Exercise 16

Prove the formal proofs 1 and 2 from the examples above using both truth tables and three-column proofs techniques.

Exercise 17

Think of a proposition and try to prove it by means of a formal proof.

2.3.4. Mathematical induction

Definition 19

Recursive functions are functions that refer to themselves. We have the following properties for such functions:

1. A simple base case (or cases) - a terminating case that returns a value without using recursion
2. A set of rules that reduce the other cases towards the base case

Definition 20

Mathematical induction is a proof method that is used to prove that a predicate $P(n)$ is true for all natural numbers n. It consists of proving two parts: a base case and an inductive step.

1. For the **base** case we need to show that what we want to prove $P(n)$ is true for some starting value k, which is usually zero.
2. For the **inductive** step, we need to prove that $P(n) \to P(n+1)$, that is, if we assume that $P(n)$ is true, then $P(n+1)$ must follow as a consequence.

After proving the two parts, we can conclude that $P(n)$ holds for all natural numbers. The formula that we need to prove is $P(0) \land (P(n) \to P(n+1))$.

To understand why mathematical induction works, as an example it is best to visualize dominoes arranged in a sequence. If we push the first domino, it will push the second, which will push the third, and so on to infinity. That is, if we position the dominoes such that if one falls it will push the next one, i.e. $P(n)$ implies $P(n+1)$, and we push the first one $P(0)$, then all the dominoes will fall, i.e. $P(n)$ is true in general.

Definition 21

We are given this recursive definition for adding numbers:

1. Zero is a left identity for addition, that is $n = 0 + n$
2. $S(m) + n = S(m + n)$, where S is the successor function, that is $S(0) = 1, S(1) = 2$, etc.

For example, in order to prove that $\forall n, n + 0 = n$ in the system of Peano's axioms, we can proceed by induction (which is an axiom in this system). For the base case, we have that $0 + 0 = 0$, which is true (by definition of adding numbers, for $n = 0$). For the inductive step, we first assume that $n + 0 = n$ is true, and prove that $S(n) + 0 = S(n)$. By definition of addition, we have $S(n) + 0 = S(n + 0)$. If we use the inductive hypothesis we have $S(n + 0) = S(n)$, which is what we needed to show.

3. Type theory

Some type theories can serve as an alternative foundation of mathematics, as opposed to standard set theory. One such well-known type theory is Martin-Löf's intuitionistic theory of types, which is an extension of Alonzo Church's simply-typed λ-calculus. Before we begin working with Idris, we will get familiar with these theories, upon which Idris is built as a language.

Definition 1

Type theory is defined as a class of formal systems. In these theories, every object is joined with a type, and operations upon these objects are constrained by the joined types. In order to say that x is of type X, we denote $x : \text{X}$. Functions are a primitive concept in type theory[14].

For example, with $1 : \text{Nat}, 2 : \text{Nat}$ we can say that 1 and 2 are of type Nat, that is natural numbers. An operation (function) $+ : \text{Nat} \to \text{Nat} \to \text{Nat}$ is interpreted as a function which takes two objects of type Nat and returns an object of type Nat.

Definition 2

In type theory, a type constructor is a function that builds new types from old ones. This function accepts types as parameters and as a result, it returns a new type.

Idris supports algebraic data types. These data types are a kind of complex types, that is, types constructed by combining other types. Two classes of algebraic types are **product types** and **sum types**.

[14] Unlike in set theory, where they are defined in terms of relations.

Definition 3

Algebraic data types are types where we can additionally specify the form for each of the elements. They are called “algebraic” in the sense that the types are constructed using algebraic operations. The algebra here is sum and product:

1. Sum (union) is an alternation. It is denoted as A | B and it means that a constructed value is either of type A or B
2. Product is a combination. It is denoted as A B and it means that a constructed value is a pair where the first element is of type A, and the second element is of type B

To understand the algebra they capture, we denote with $|\text{A}|$ the number of possible values of type A. When we create an algebraic sum we have $|\text{A} \mid \text{B}| = |\text{A}| + |\text{B}|$. Similarly, for algebraic product we have $|\text{A B}| = |\text{A}| * |\text{B}|$.

As an example, we can assume that we have two types: Nat for natural numbers, and Real for real numbers. Using sum (union) we can construct a new type Nat | Real. Valid values of this type are 1 : Nat | Real, 3.14 : Nat | Real, etc. Using product we can construct a new type Nat Real. Valid values of this type are $1\ 1.5$: Nat Real, $2\ 3.14$: Nat Real, etc. With this, sums and products can be combined and thus more complex data structures can be defined.

The type Bool has two possible values: $True$ and $False$. Thus, $|\text{Bool}| = 2$. The type Unit (equivalent to $()$) has one possible value: $Unit$. We can now form a sum type Bool | Unit which has length 3 with values $True, False, Unit$. Additionally, the product type Bool Unit has length 2 with values $True\ Unit, False\ Unit$.

Besides the sum and product, there is another important operation called **exponentiation**. This corresponds to functions, so a type $a \to b$ has $|b|^{|a|}$ possible values. In section 3.3 we will see how this algebra can be generalized.

Finally, Idris supports dependent types[15]. These kinds of types are so powerful, they can encode most properties of programs and with their help, Idris can prove

[15] Dependent types allow proofs of statements involving first-order predicates, compared to simple types that correspond to propositional logic. While useful (since we can check whether an expression fulfills a given condition at compile-time), dependent types add complexity to a type system. In order to calculate type “equality” of dependent types, computations are necessary. If we allow any values for dependent types, then solving equality of a type may involve deciding whether two programs produce the same result. Thus, the check may become undecidable.

invariants at compile-time. As we will see in section 4.2 types also allow us to encode mathematical proofs, which brings computer programs closer to mathematical proofs. As a consequence, this allows us to prove properties (e.g. specifications) about our software[16].

Why are types useful?

Russell’s paradox (per the mathematician Bertrand Russell) states the following: In a village in which there is only one barber, there is a rule according to which the barber shaves everyone who don’t shave themselves, and no-one else. Now, who shaves the barber? Suppose the barber shaves himself. Then, he’s one of those who shave themselves, but the barber shaves only those who do not shave themselves, which is a contradiction. Alternatively, if we assume that the barber does not shave himself, then he is in the group of people whom which the barber shaves, which again is a contradiction. Apparently then the barber does not shave himself, but he also doesn’t *not* shave himself - a paradox.

Some set theories are affected by Russell’s paradox. As a response to this, between 1902 and 1908, Bertrand Russell himself proposed different type theories as an attempt to resolve the issue. By joining types to values, we avoid the paradox because in this theory every set is defined as having elements from a distinct type, for example, Type 1. Elements from Type 1 can be included in a different set, say, elements of Type 2, and so forth. Thus, the paradox is no longer an issue since the set of elements of Type 1 cannot be contained in their own set, since the types do not match. In a way, we’re adding hierarchy to sets in order to resolve the issue of “self-referential” sets. This is also the case with Idris, where we have that Type : Type 1 : Type 2, etc.

Thus, for Russell’s paradox specifically, if we set the type of a person to be P, then the list of people would be of type List P. However, there is no way to express $\{P\}$ such that $P \in P$, since List P only contains elements of type P, and not List P.

[16] This is what makes Idris a so-called proof assistant. In general, Idris combines a lot of functionalities from mainstream languages (Java, C, C++) and some functionalities from proof assistants, which further blurs the line between these two kinds of software.

3.1. Lambda calculus

Lambda calculus is a formal system for expressing computation[17]. The grammar rules are divided in two parts: function abstraction and function application. Function abstraction defines what a function does, and function application "computes" a function. For example $f(x) = x + 1$ is a function abstraction and $f(3)$ is a function application. The equality sign $=$ is replaced with a dot, and instead of writing $f(x)$ we write λx. To represent $f(x) = x + 1$ we write $\lambda x.x + 1$. Parentheses allow us to specify the order of evaluation.

Definition 4

The set of symbols for the lambda calculus is defined as:

1. There are variables $v_1, v_2, \ldots$
2. There are only two abstract symbols: . and λ
3. There are parentheses: (and)

The set of grammar rules Λ for well-formed expressions is defined as:

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$ (rule of abstraction)
3. If $M, N \in \Lambda$, then $(M\ N) \in \Lambda$ (rule of application)

Some examples of well-formed expressions are $\lambda f\ x.f\ x$ and $\lambda f\ x.f\ (f\ x)$. In fact, we can encode numbers this way. The first expression can be thought of as the number one, and the second as the number two. In other words, the number 1 is defined roughly as $f(x)$, and 2 as $f(f(x))$. Note that f and x do not have special definitions, they are abstract objects. This encoding is known as the Church encoding. Operations on numbers (plus, minus, etc) and other data such as Booleans and lists can also be encoded similarly.

[17] A *Turing machine* is a very simple abstract machine designed to capture our intuitive understanding of *computation* in the most general sense. Any formal system that can simulate a Turing machine, and thus also perform arbitrary computations, is called *Turing complete*.

Note that $\lambda f\ x.f\ x$ is shorthand for $\lambda f.\lambda x.f\ x$.

Exercise 1

Convince yourself that the expression $\lambda f\ x.f\ x$ is a well-formed expression by writing down each one of the grammar rules used.

3.1.1. Term reduction

Every variable in a lambda expression can be characterized as either *free* or *bound* in that expression.

Definition 5

A variable in a lambda expression is called *free* if it does not appear inside at least one lambda body where it is found in the abstraction. Alternatively, if it does appear inside at least one lambda body, then the variable is *bound* at the innermost such lambda abstraction.

This definition of "bound" corresponds roughly to the concept of *scope* in many programming languages. Lambda expressions introduce a new scope in which their argument variables are bound.

For example, in the expression $\lambda y.x\ y$ we have that y is a bound variable, and x is a free one. Variable binding in lambda calculus is subtle but important, so let's see some trickier examples.

1. In $x(\lambda x.x)$, the leftmost x is free, while the rightmost x is bound by the lambda
2. In $(\lambda x.x)(\lambda x.x)$, both occurrences of x are bound; the first at the left lambda, and the second at the right lambda
3. In $\lambda x.(\lambda x.x)$ the sole occurrence of x is certainly bound. Now there are two potential "binding sites" - the inner lambda and the outer lambda. Given a choice like this, we always say the variable is bound at the innermost lambda

The distinction between free and bound variables becomes important when we ask whether two different lambda expressions are "equal". For instance, consider the two expressions $\lambda x.x$ and $\lambda y.y$. Syntactically these are not the same; they use different characters for the variable. But semantically they are identical because in lambda calculus variables bound by a lambda are "dummy" variables whose exact names are not important. When two lambda expressions differ only by a consistent renaming of the bound variables like this we say they are *alpha equivalent.*

There are two other useful notions of semantic equivalence for lambda expressions: beta and eta equivalence.

Definition 6

The rules of terms reduction (inference rules) allow us to compute (simplify) lambda expressions. There are three types of reduction:

1. α - (alpha) reduction: Renaming bound variables
2. β - (beta) reduction: Applying arguments to functions
3. η - (eta) reduction: Two functions are "equal" iff they return the same result for all arguments

For example, for the expression $(\lambda x.f\ x)\ y$, we can use alpha reduction to get to $(\lambda z.f\ z)\ y$, by changing x to z. Using beta reduction, the expression can further be reduced to just $f\ y$, since we "consumed" the z by removing it from the abstraction and wherever it occurred in the body we just replaced it with what was applied to it, that is y. Finally, with eta reduction, we can rewrite $(\lambda x.f\ x)$ to just f, since they are equivalent.

Given these rules, we can define the successor function as SUCC $= \lambda n\ f\ x\ .f\ (n\ f\ x)$. Now we can try to apply 1 to SUCC:

1. Evaluating SUCC $1 =$
2. Substitute the very own definitions of SUCC and 1: $(\lambda n\ f\ x.f\ (n\ f\ x))\ (\lambda f\ x.f\ x) =$
3. Apply 1 to SUCC i.e. "consume" n by beta reduction: $\lambda f\ x.f\ ((\lambda f\ x.f\ x)\ f\ x) =$
4. Finally, apply f and x to a function that accepts f and x (which is just the body of the abstraction): $\lambda f\ x.f\ (f\ x) = 2$

Definition 7

A fixed-point combinator is any function that satisfies the equation fix $f = f$ (fix f).

One example of such a combinator is Y $= \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$. This definition satisfies Y $f = f$ (Y f). This combinator allows for recursion in lambda calculus. Since it is impossible to refer to the function within its body, recursion can only be achieved by applying parameters to a function which is what this combinator does.

Exercise 2

Evaluate SUCC 2 to find out the definition of number 3.

Exercise 3

Come up with your own functions that operate on the Church numerals. It can be as simple as returning the same number, or a constant one.

3.2. Lambda calculus with types

So far we've been discussing the *untyped* lambda calculus, but it is possible to augment the rules of lambda calculus so that variables are *typed*. This makes it possible to add an additional statically checked layer of semantics to a lambda expression so we can ensure that values are used in a consistent and meaningful way. There are several ways to add types to lambda calculus, and our goal is to approach the full *dependent* type system of Idris. As a stepping stone, we first consider *simple* types.

Definition 8

Simply typed lambda calculus is a type theory that adds types to lambda calculus. It joins the system with a unique type constructor $\rightarrow$ which constructs types for functions. The formal definition and the set of lambda expressions are similar to that of lambda calculus, with the addition of types.

The set of symbols for this system is defined as:

1. There are variables $v_1, v_2, \ldots$
2. There are only two abstract symbols: . and λ
3. There are parentheses: (and)

The set of grammar rules Λ for well-formed expressions is defined as:

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$ (rule of abstraction)
3. If $M, N \in \Lambda$, then $(M\ N) \in \Lambda$ (rule of application)
4. If x is a variable, T is a type, and $M \in \Lambda$, then $(\lambda x : \mathrm{T}.M) \in \Lambda$
5. If x is a variable and T is a type, then $x : \mathrm{T} \in \Lambda$

There is a single type constructor:

1. For some type A, the type constructor T is defined as A | T $\rightarrow$ T

That is, an expression in this system can additionally be an abstraction with x having joined a type (rule 4) or an expression of a variable having joined a type T (rule 5), where our type constructor is a sum type and it says that we either have primitive types or a way to form new types. In an attempt to re-define Church numerals and the successor function we have to be careful as the types of these definitions have to match. Let's recall the Church numerals:

1. Number 1, i.e. $\lambda f\ x.f\ x$
2. Number 2, i.e. $\lambda f\ x.f\ (f\ x)$

Given the definition of 1, its type must have the form $(\mathrm{a} \to \mathrm{b}) \to \mathrm{a} \to \mathrm{b}$ for some values a and b. We are expecting to be able to apply f to x, and so if $x : \mathrm{a}$ then $f : \mathrm{a} \to \mathrm{b}$ in order for our types to match correctly. With similar reasoning, we have the same type for 2. At this point, we have the type of $(\mathrm{a} \to \mathrm{b}) \to \mathrm{a} \to \mathrm{b}$. Finally, with the given definition of 2, we can note that expressions of type b need to be able to be passed to functions of type $\mathrm{a} \to \mathrm{b}$, since the result of applying f to x serves as the argument of f. The most general way for that to be true is if $a = b$. As a result we have the type $(\mathrm{a} \to \mathrm{a}) \to \mathrm{a} \to \mathrm{a}$. We can denote this type definition to be Nat. We have the following for our numbers:

1. Number 1 becomes $\lambda[f : (\mathrm{a} \to \mathrm{a})]\ [x : \mathrm{a}].f\ x : \mathrm{Nat}$
2. Number 2 becomes $\lambda[f : (\mathrm{a} \to \mathrm{a})]\ [x : \mathrm{a}].f\ (f\ x) : \mathrm{Nat}$

The (typed) successor function is: $\mathrm{SUCC} = \lambda[n : \mathrm{Nat}]\ [f : (\mathrm{a} \to \mathrm{a})]\ [x : \mathrm{a}].f\ (n\ f\ x) : \mathrm{Nat} \to \mathrm{Nat}$.

Simply typed lambda calculus sits in a sweet spot on the spectrum of type systems. It is powerful enough to do useful work, but also simple enough to have strong properties. Simple types have limitations when compared to full dependent types, discussed in the next section, but their great trade-off is the existence of a full *inference algorithm*. The strategy we used above to determine the type of a lambda expression from the bottom up is the core of the widely used Hindley-Damas-Milner algorithm for type inference, which can automatically *infer* the simple type of a lambda expression without requiring any explicit type annotations from the programmer.

Fixed-point combinators do not exist in the simply-typed lambda calculus[18]. To see why, consider the function $\mathrm{fix} = \lambda[f : \mathrm{a} \to \mathrm{a}] : \mathrm{a}$. We can apply fix to some element $x : \mathrm{a} \to \mathrm{a}$. Thus, $\mathrm{fix}\ x : \mathrm{a}$, but $x = \mathrm{fix}\ x$ is a type error because the infinite type $\mathrm{a} \to \mathrm{a} = \mathrm{a}$ cannot be matched.

Exercise 4

Come up with a definition of the typed number 3.

[18] For this reason, the typed lambda calculus is not Turing complete, while the untyped lambda calculus is. Fixed-point combinators provide flexibility, but that has its drawbacks. They can be non-terminating - loop indefinitely without producing an answer. While non-termination has its uses for software (e.g. a program keeps running until we choose to close it), termination is important for mathematical proofs as we will see in section 4.2.

Exercise 5

Apply the typed 1 to SUCC and confirm that the result is 2. Make sure you confirm that the types also match in the process of evaluation.

Exercise 6

In Exercise 3 you were asked to come up with a function. Try to figure out the type of this function or, if not applicable, come up with a new function and then figure out its type using the reasoning above.

3.3. Dependent types

In the simply-typed lambda calculus, *values* and *types* are fundamentally different kinds of things that are related only by the "has type" predicate, :. Values are allowed to depend on values - these are lambda abstractions. And types are allowed to depend on types - these are arrow types. But types are not allowed to depend on values. A *dependent typing* system lifts this restriction.

Definition 9

Dependent types are types that depend on values.

A list of numbers is a type (List Nat, for example). However, a list of numbers whose length is bounded by some constant, or whose entries are increasing, is a dependent type.

Definition 10

A dependent product type is a collection of types $B : \mathrm{A} \to U$ where for each element $a : \mathrm{A}$, there's an assigned type $B(a) : U$, where U is a universe of types[19]. We say that $B(a)$ varies with a. It is denoted as $\Pi(x : \mathrm{A}), B(x)$ or $\prod_{x \,:\, \mathrm{A}} B(x)$.

[19] Collections, in general, are considered to be subcollections of some large universal collection, also called the universe. Depending on the context, the definition of this universe will vary.

This definition might seem a bit scary and tricky to grasp, but it really is simple, and it is best to see it in action through the following example:

1. Our universe of types contains all possible types. For example, Type, Nat, etc, so $U = \{\text{Type}, \text{Nat}, \text{List n}, \dots\}$
2. Our collection of types of interest is List n, which represents a list of n elements. That is, $A = \{\text{List n}\}$

The definition states that in the universe U, there exists a function $B(\text{n}) = \text{List n}$. B is the collection of functions which given a number n, will return a list of n numbers. For example, we have the following lists:

1. List of 1 element: $[1] : B(1)$, that is $[1] : \text{List } 1$
2. List of 2 elements: $[1, 2] : \text{List } 2$
3. List of n elements: $[1, 2, \dots, n] : \text{List n}$

In general, we have a function that takes an n and produces a List n, that is, $f : \Pi(x : \text{Nat}), \text{n} \rightarrow \text{List n}$ or simply $f : \text{n} \rightarrow \text{List n}$, where the possible types for it are $f : 1 \rightarrow \text{List } 1$ and $f : 2 \rightarrow \text{List } 2$, etc. We've just constructed our first dependent type!

Definition 11

A dependent sum type can be used to represent indexed pairs, where the type of the second element depends on the type of the first element. That is, if we have $a : \text{A}$ and $b : B(\text{a})$, then this makes a sum type. We denote it as $\Sigma(x : \text{A}), B(x)$ or $\sum_{x\,:\,\text{A}} B(x)$.

For example, if we set $A = \text{Nat}$, and $B(\text{a}) = \text{List a}$, then we form the dependent sum type $\Sigma(x : \text{Nat}), \text{List x}$. Possible types for it are $(1, \text{List } 1)$ or $(2, \text{List } 2)$, etc. For example, we can construct the following pairs: $(1, [1]), (2, [1, 2]), (3, [1, 2, 3])$, etc.

Dependent types generalize product and exponentiation. Namely, Σ (multiplication) is a generalization of the product type where the type of the second element depends on the first element, and Π (exponentiation) is a generalization of the exponentiation type where the resulting type of a function depends on its input.

Exercise 7

Think of a way to construct a different dependent product type and express it by using the reasoning above.

Exercise 8

Think of a way to construct a different dependent sum type and express it using the reasoning above.

3.4. Intuitionistic theory of types

The core "construct" in Idris are types. As we've seen, foundations are based on type theory. As we've also seen, in classical mathematical logic we have sets and propositions, according to set theory.

The intuitionistic theory of types (or constructive type theory) offers an alternative foundation to mathematics. This theory was introduced by Martin-Löf, a Swedish mathematician in 1972. It is based on the isomorphism (or "equality") that propositions are types.

Proving a theorem in this system consists of constructing[20] (or providing evidence for) a particular object. If we want to prove something about a type A and we know that a : A, then a is one proof for A. Note how we say one proof, because there can be many other elements of type A.

Propositions can also be defined through types. For example in order to prove that $4 = 4$, we need to find an object x of type 4 = 4, that is x : 4 = 4. One such object is $refl$ (which can be thought of as an axiom), which stands for reflexivity, which states that $x = x$ for all x.

[20] As a consequence that we need to construct an object as evidence in order to prove something, the law of excluded middle $P \vee \neg P$ is not valid in this logic, whereas in classical mathematical logic this is taken as an axiom. For some propositions, for example, P is an odd number or not, there are proofs that we can provide. However, for some propositions this is impossible, for example, P is a program that halts or not. Unlike classical mathematical logic, in this logic, the law of excluded middle does not exist due to the undecidability problem.

One thing worth noting is that in Idris there are “two” types of truths: Bool and Type. Even though there is some similarity (in terms of proofs), in Idris they are fundamentally different. The type Bool can have a value of $True$ or $False$, while the type Type is either provable or not provable[21].

This system is useful since with the use of computer algorithms we can find a constructive proof for some object (assuming it exists). As a consequence, this is why it can be considered as a way to make a programming language act like a proof-assistant.

[21] It is provable in case we can construct an object of such type, and not provable otherwise.

Definition 12

The set of grammar rules Λ for well-formed expressions is defined as:

1. $s : \mathrm{Type} \in \Lambda$ means that s is a well-formed type
2. $t : \mathrm{s} \in \Lambda$ means that t is a well-formed expression of type s
3. $\mathrm{s} = \mathrm{t} \in \Lambda$ means that s and t are the same type
4. $t = u : \mathrm{s} \in \Lambda$ means that t and u are equal expressions of type s

The type constructors are:

1. Π types and Σ types, as we've discussed them earlier
2. Finite types, for example the nullary (empty) type 0 or $\bot$, the unary type 1 or $\top$, and the boolean type 2
3. The equality type, where for given $a, b : \mathrm{A}$, the expression $a = b$ represents proof of equality. There is a canonical element $a = a$, that is, an "axiom" for the reflexivity proof: $refl : \Pi(a : \mathrm{A})\ a = a$
4. Inductive (or recursive) types. This way we can implement a special form of recursion - one that always terminates, and we will see the importance of this with total functions. Additionally we can implement product and sum types, which encode conjunction and disjunction respectively

The inference rules are:

1. The rule of type equality which states that if an object is of a type A, and there is another type B equal to A, then that object is of type B: $(a : \mathrm{A}, \mathrm{A} = \mathrm{B}) \to (a : \mathrm{B})$

There are other inference rules, for example introduction and elimination. We will show an example of using these rules in section 4.2.

As an example, for well-formed expressions rule 1 says that we can form an expression such that an object inhabits the type Type, so an example of a well-formed expression is $1 : \mathrm{Nat}$, per rule 2, and $\mathrm{Nat} : \mathrm{Type}$ per rule 1.

A valid type as per the fourth rule of type constructors is the definition of natural numbers Nat = Z | S Nat. Some valid values are Z : Nat, S Z : Nat, etc.

Exercise 9

We used rule 1 and rule 2 in the example earlier. Try to come up with different ways to use some of the rules.

Exercise 10

Combine the use of rules along with the connectives described earlier and try to come up with a recursive type and then construct some new objects from it.

3.4.1. Intuitionistic logic

Definition 13

A constructive proof proves the existence of a mathematical object by creating or constructing the object itself. This is contrary to non-constructive proofs which prove the existence of objects without giving a concrete example.

Definition 14

Intuitionistic logic, also known as constructive logic, is a type of logic that is different than classical logic in that it "works" with the notion of constructive proof.

Definition 15

The BHK (Brouwer-Heyting-Kolmogorov) interpretation is a mapping of intuitionistic logic to classical mathematical logic, namely:

1. A proof of $P \wedge Q$ is a product type A B, where a is a proof of (or, object of type) P and b is a proof of Q
2. A proof of $P \vee Q$ is a product type A B, where a is 0 and b is a proof of P, or a is 1 and b is a proof of Q
3. A proof of $P \to Q$ is a function f that converts a proof of P to a proof of Q
4. A proof of $\exists x \in S : f(x)$ is a pair A B where a is an element of S, and b is a proof of $f(a)$ (dependent sum types)
5. A proof of $\forall x \in S : f(x)$ is a function f that converts an element a from S to a proof of $f(a)$ (dependent product types)
6. A proof of $\neg P$ is defined as $P \to \bot$, that is, the proof is a function f that converts a proof of P to a proof of $\bot$
7. There is no proof of $\bot$

For example, to prove distributivity of $\wedge$ with respect to $\vee$, that is, $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$, we need to construct a proof for the function of type f : P (Q | R) → P Q | P R. That is, a function that takes a product type of P and sum type of Q and R, and returns a sum type of product P and Q, and product P and R. Here's the function that accomplishes that:

```
f (x, left y)  = left (x, y)
f (x, left y') = right (x, y')
```

This notation (which is pretty similar to how we would write it in Idris) uses `(x, y)` to denote product type, that is, extract values from a product-type pair, and `left` and `right` to denote value constructors for sum type in order to extract values from a sum-type pair. In the next chapter we will introduce Idris and its syntax.

Exercise 11

Try to use some of the proofs in the earlier chapters as motivation and work them out by using intuitionistic logic.

4. Programming in Idris

In this chapter, we will introduce Idris' syntax, by defining functions and types.

Depending on what level of abstraction we are working on, types and proofs can give us a kind of security based on some truths we take for granted (axioms). In fact, this is how we develop code on a daily basis, as software engineers. We have a list of axioms, for example, a `foreach` loop in a programming language, and starting from it we build abstractions, expecting this `foreach` to behave in a certain way. However, this security is not always easy to achieve. For example, consider a scenario where we have a button that is supposed to download a PDF document. In order to prove that it works as expected, we must first pick the abstraction level we will be working on, and then proceed by defining software requirements (what is a PDF, what is a download). So, we first have to define our **specifications**, and then we can proceed with proving correctness.

Idris, even though a research language, still has its own uses. It is a Turing complete language, which means that it has the same expressive power as other programming languages, for example, Java, or C++.

To start with, Idris can be downloaded and installed via www.idris-lang.org/download.

4.1. Basic syntax and definitions

There are two working modes in Idris: REPL (read-evaluate-print-loop) i.e. interactive mode, and compilation of code. We will mostly work in the interactive mode in this chapter.

You can copy any of the example codes to some file, say, `test.idr` and launch Idris in REPL mode by writing `idris test.idr`. This will allow you to interact with Idris given the definitions in the file. If you change the contents of the file, that is, update the definitions, you can use the command `:r` while in REPL mode to reload the definitions.

4.1.1. Defining functions

Let's begin by defining a simple function.

```
add_1 : Nat -> Nat
add_1 x = x + 1
```

With the code above we're defining a function $f(x) = x+1$, where x is natural number and $f(x)$ (the result) is also a natural number[22]. The first line `add_1 : Nat -> Nat` is called a *type signature*, and specifies the type of our function; in this case a function that takes a natural number and returns a natural number. The second line `add_1 x = x + 1` is the definition of the function, which states that if `add_1` is called with a number `x`, the result would be `x + 1`. As can be seen by the example, every function has to be provided a type definition. We can interact as follows in the REPL mode:

```
Idris> add_1 5
6 : Nat
```

Idris responds to us that as a result, we get `6`, which is of type `Nat`. Constants are defined similarly; we can think of them as functions with no arguments.

```
number_1 : Nat
number_1 = 1
```

As in Haskell, we can use pattern matching. What happens during this phase is Idris does a check (match) against the definitions of the function and uses the definition of the function that matches the value.

[22] It is worth noting that in Haskell we have types and kinds. Kinds are similar to types, that is, they are defined as one level above types in simply-typed lambda calculus. For example, types such as `Nat` have a kind `Nat :: *` and it's stated that `Nat` is of kind `*`. Types such as `Nat -> Nat` have a kind of `* -> *`. Since in Idris the types are first-class citizens, there is no distinction between types and kinds.

```
is_it_zero : Nat -> Bool
is_it_zero Z = True
is_it_zero x = False
```

We've just defined a function that accepts a natural number and returns a boolean value (`True` or `False`). On the first line, we specify its type. On the second line, we pattern match against the input `Z` and return `True` in that case. On the third line, we pattern match against the input `x` (which is all remaining inputs except `Z`). In this case, we return `False`. The code above, depending on the input, will branch the computation to the corresponding definition. Note that `Z` corresponds to `0` for the type `Nat`.

Exercise 1

Write a function `my_identity` which accepts a natural number and returns the same number.

Exercise 2

Write a function `five_if_zero` which accepts a natural number and returns 5 when called with an argument of 0, otherwise returns the same number. For example, `five_if_zero 0` should return 5. `five_if_zero 123` should return 123.

4.1.2. Defining and inferring types

In Idris, types are first-class citizens. This means that types can be computed and passed to other functions. We define new types by using the keyword `data`. All concrete types (like `Nat`) and type constructors (like `List`) begin with an uppercase letter, while lowercase letters are reserved for polymorphic types[23]. There are a couple of ways to define types. One example is by using the so-called `Haskell98` syntax:

[23] A polymorphic type can accept additional types as arguments, which are either defined by the programmer or primitive ones.

```
data A a b = A_inst a b
```

This will create a polymorphic type that accepts two type arguments, `a` and `b`. Valid constructed types are `A Nat Bool`, `A Nat Nat`, etc. `A` is a type constructor (a function that returns a type) and `A_inst`[24] is a value constructor (a function that returns a value of type `A a b`).

```
Idris> A_inst True True
A_inst True True : A Bool Bool
Idris> A_inst Z True
A_inst 0 True : A Nat Bool
```

Note how the type changes depending on what values we pass to the constructor, due to polymorphism.

Here's an equivalent definition by using the `GADT` (Generalized Algebraic Data Types) syntax:

```
data A : Type -> Type -> Type where
    A_inst : a -> b -> A a b
```

This is equivalent to the following definition, where we define an empty data structure along with an axiom for the value constructor:

```
data A : Type -> Type -> Type
postulate A_inst : a -> b -> A a b
```

With the `postulate` keyword we can define axioms which are functions that satisfy the types without giving an actual argument for construction. With the command `:t` we can check the type of an expression, so as a result, we get:

[24] The value and type constructors must be named differently since types and values are at the same level in Idris.

```
Idris> :t A
A : Type -> Type -> Type
Idris> :t A_inst
A_inst : a -> b -> A a b
```

That is, we show the type definitions for both the newly-defined type and its value constructor. Note how we created a product type here. Idris has a built-in[25] notion of pairs, which is a data type that can be defined in terms of products. For example, `(1, 2)` is one pair. We can also define tuples with `(1, "Hi", True)`, which is equivalent to `(1, ("Hi", True))`, i.e. a pair where the first element is a number, and the second element is a pair. Note that the types `(a, b) -> c` (curried) and `a -> b -> c` (uncurried) represent the same thing.

Analogously, if we want to create a sum type, we could do the following:

```
data B a b = B_inst_left a | B_inst_right b
```

This is equivalent to:

```
data B : Type -> Type -> Type where
    B_inst_left : a -> B a b
    B_inst_right : b -> B a b
```

With either of these definitions in scope, we get:

```
Idris> :t B
B : Type -> Type -> Type
Idris> :t B_inst_left
B_inst_left : a -> B a b
Idris> :t B_inst_right
B_inst_right : b -> B a b
```

The functions `B_inst_left` and `B_inst_right` are also known as *introduction rules*. For extracting values from data types such as `B a b`, also known as *elimination rules*, we can use pattern matching[26]. As an example, to extract `a` from `B a b`, we can use

[25] By built-in, we usually mean it's part of Idris' library. We can always implement it ourselves if we need to.

[26] Although product and sum types are very general, due to polymorphism, we can say something very specific about the structure of their values. For instance, suppose we've defined a type like so: `data C a b c = C_left a | C_right (b,c)`. A value of type `C` can only come into existence in one of two ways: as a value of the form `C_left x` for a value `x : a`, or as a value of the form `C_right (y,z)` for values `y : b` and `z : c`.

the following function:

```
f : B a b -> a
f (B_inst_left a) = a
```

Note how we used the data type at the function type level and the value constructor in the function definition to pattern match against.

Natural numbers can be defined as `data Nat = Z | S Nat`, where we either have a zero or a successor of a natural number. Note how this type is not polymorphic (it doesn't accept any variables after the type name). Natural numbers are built-in as a type in Idris.

We can use the operator == to compare two numbers. Note that == is still a function, but it's an infix one. This means that unlike other functions that we define which are prefix, i.e. start at the beginning of the expression, == needs to appear between the parameters. For example:

```
Idris> 1 == 1
True : Bool
```

Idris has several built-in primitive types, including: `Bool`, `Char`, `List`, `String` (list of characters), `Integer`, etc. The only value constructors for the type `Bool` are `True` and `False`. The difference between a `Nat` and an `Integer` is that `Integer` can also contain negative values. Here are some examples of interacting with these types:

```
Idris> "Test"
"Test" : String
Idris> '!'
'!' : Char
Idris> 1
1 : Integer
Idris> '1'
'1' : Char
Idris> [1, 2, 3]
[1, 2, 3] : List Integer
```

By using the `:doc` command, we can get detailed information about a data type:

```
Idris> :doc Nat
Data type Prelude.Nat.Nat : Type
    Natural numbers: unbounded, unsigned integers which can be pattern
    matched.

Constructors:
    Z : Nat
        Zero

    S : Nat -> Nat
        Successor
```

In order to make Idris infer the necessary type of the function that needs to be built, we can take advantage of holes. A hole is any variable that starts with a question mark. For example, if we have the following definition:

```
test : Bool -> Bool
test p = ?hole1
```

We can now ask Idris to tell us the type of `hole1`, that is, with `:t hole1` we can see that Idris inferred that the specific result is expected to be of type `Bool`. This is useful because it allows us to write programs incrementally (piece by piece) instead of constructing the program all at once.

Exercise 3

Define your own custom type. One example is `data Person = PersonInst String Nat`, where `String` represents the Person's name and `Nat` represents the Person's age. Use the constructor to generate some objects of that type. Afterward, use `:t` to check the types of the type and value constructors.

Exercise 4

Come up with a function that works with your custom type (for example, it extracts some value) by pattern matching against its value constructor(s).

Exercise 5

Repeat Exercise 4 and use holes to check the types of the expressions used in your function definition.

4.1.3. Anonymous lambda functions

With the syntax `let X in Y` we're defining a set of variables `X` which are only visible in the body of `Y`. As an example, here is one way to use this syntax:

```
Idris> let f = 1 in f
1 : Integer
```

Alternatively, the REPL has a command `:let` that allows us to set a variable without evaluating it right away, that can be used at a later point:

```
Idris> :let f = 1
Idris> f
1 : Integer
```

What's the difference between `let` and `:let`?

`let` is an extension of the lambda calculus syntax. The expression `let x = y in e` is equivalent to `(\x . e) y`. It is useful since it makes it easy to shorten a lambda expression by factoring out common subexpressions.

`:let` is different in that it is just used to bind new names at the top level of an interactive Idris session.

Lambda (anonymous) functions are defined with the syntax `\a, b, ..., n => ...`. For example:

```
Idris> let addThree = (\x, y, z => x + y + z) in addThree 1 2 3
6 : Integer
```

With the example above, we defined a function `addThree` that accepts three parameters and as a result it sums them. However, if we do not pass all parameters to a function, it will result in:

```
Idris> let addThree = (\x, y, z => x + y + z) in addThree 1 2
\z => prim__addBigInt 3 z : Integer -> Integer
```

We can see (from the type) that as a result, we get another function.

Definition 1

Currying is a concept that allows us to evaluate a function with multiple parameters as a sequence of functions, each having a single parameter.

Function application in Idris is left-associative (just like in lambda calculus), which means that if we try to evaluate `addThree 1 2 3`, then it will be evaluated as `(((addThree 1) 2) 3)`. A combination of left-associative functions and currying (i.e. partial evaluation of function) is what allows us to write `addThree 1 2 3`, which is much more readable.

Arrow types are right-associative (just like in lambda calculus), which means that `addThree 1 : a -> a -> a` is equivalent to `addThree 1 : (a -> (a -> a))`. If we had written a type of `(a -> a) -> a` instead, then this function would accept as its first parameter a function that takes an `a` and returns an `a`, and then the original function also returns an `a`. This is how we can define higher-order functions which we will discuss later. Note how `a` starts with a lowercase letter, thus it is a polymorphic type.

The `if...then...else` syntax is defined as follows:

```
Idris> if 1 == 1 then 'a' else 'b'
'a' : Char
Idris> if 2 == 1 then 'a' else 'b'
'b' : Char
```

Exercise 6

Write a lambda function that returns `True` if the parameter passed to it is 42 and `False` otherwise.

4.1.4. Recursive functions

By Definition 19 of chapter 2, recursive functions are those functions that refer to themselves. One example is the function `even : Nat -> Bool`, a function that checks if a natural number is even or not. It can be defined as follows:

```
even : Nat -> Bool
even Z     = True
even (S k) = not (even k)
```

The definition states that `0` is an even number, and that `n + 1` is even or not depending on the parity (evenness) of `n`. As a result, we get:

```
Idris> even 3
False : Bool
Idris> even 4
True : Bool
Idris> even 5
False : Bool
```

The way `even 5` unfolds in Idris is as follows:

```
even 5 =
not (even 4) =
not (not (even 3)) =
not (not (not (even 2))) =
not (not (not (not (even 1)))) =
not (not (not (not (not (even 0))))) =
not (not (not (not (not True)))) =
not (not (not (not False))) =
not (not (not True)) =
not (not False) =
not True =
False
```

We see how this exhibits a recursive behavior since the recursive cases were reduced to the base case in an attempt to get a result. With this example, we can see the power of recursion and how it allows us to process values in a repeating manner.

A recursive function can generate an **iterative** or a **recursive** process:

1. An iterative process[27] (tail recursion) is a process where the return value at any point in computation is captured completely by its parameters
2. A recursive one, in contrast, is one where the return value is not captured at any point in computation by the parameters, and so it relies on postponed evaluations

In the example above, `even` generates a recursive process since it needs to go down to the base case, and then build its way back up to do the calculations that were postponed. Alternatively, we can rewrite `even` so that it captures the return value by introducing another variable, as such:

[27] The key idea is not that a tail-recursive function *is* an iterative loop, but that a smart enough compiler can *pretend* that it is and evaluate it using constant function stack space.

```
even : Nat -> Bool -> Bool
even Z t     = t
even (S k) t = even k (not t)
```

In this case, we can refer to the return value of the function (second parameter) at any point in the computation. As a consequence, this function generates an iterative process, since the results are captured in the parameters. Note how we brought down the base case to refer to the parameter, instead of a constant `True`. Here is how Idris evaluates `even 5 True`:

```
even 5 True =
even 4 False =
even 3 True =
even 2 False =
even 1 True =
even 0 False =
False
```

To conclude, iterative processes take fewer calculation steps and are usually more performant than recursive processes. Recursive functions combined with pattern matching are one of the most powerful tools in Idris since they allow for computation. They are also useful for proving mathematical theorems with induction, as we will see in the examples later.

Exercise 7

Factorial is defined as: $fact(n) = \begin{cases} 1, \text{if } n = 0 \\ n \cdot fact(n-1), \text{otherwise} \end{cases}$

Unfold the evaluation of $fact(5)$ on paper, and then implement it in Idris and confirm that Idris also computes the same value.

Hint: The type is `fact : Nat -> Nat` and you should pattern match against `Z` (`Nat` value constructor for 0) and `(S n)` (successor).

Exercise 8

Re-write the factorial function to generate an iterative process.

Hint: The type is `fact_iter : Nat -> Nat -> Nat` and you should pattern match against `Z acc` (number zero, and accumulator) and `(S n) acc` (successor, and accumulator).

4.1.5. Recursive data types

We can think of type constructors as functions at the type level. Taking the concept of recursion to this context yields *recursive types*.

Definition 2

A recursive data type is a data type where some of its constructors have a reference to the same data type.

We will start by defining a recursive data type, which is a data type that in the constructor refers to itself. In fact, earlier in this book we already gave a recursive definition - `Nat`. As a motivating example, we will try to define the representation of lists. For this data type, we'll use a combination of sum and product types. A list is defined as either `End` (end of the list) or `Cons` (construct), which is a value appended to another `MyList`:

```
data MyList a = Cons a (MyList a) | End
```

This means that the type `MyList` has two constructors, `End` and `Cons`. If it's `End`, then it's the end of the list (and does not accept any more values). However, if it's `Cons`, then we need to append another value (e.g. `Cons 3`), but afterward, we have to specify another value of type `MyList a` (which can be `End` or another `Cons`). This definition allows us to define a list. As an example, this is how we would represent $(1, 2)$ using our `Cons End` representation:

```
Idris> :t Cons 1 (Cons 2 End)
Cons 1 (Cons 2 End) : MyList Integer
Idris> :t Cons 'a' (Cons 'b' End)
Cons 'a' (Cons 'b' End) : MyList Char
```

Note how Idris automatically infers the polymorphic type to `MyList Integer` and `MyList Char`. In particular, these lists are *homogeneous*; all the items in a `MyList a` must have type `a`.

Here is one way of implementing the concatenation function, which given two lists, should produce a list with the elements appended:

```
add' : MyList a -> MyList a -> MyList a
add' End ys         = ys
add' (Cons x xs) ys = Cons x (add' xs ys)
```

The first line of the code says that `add'` is a function that accepts two polymorphic lists (`MyList Nat`, `MyList Char`, etc), and produces the same list as a result. The second line of the code pattern matches against the first list and when it's empty we just return the second list. The third line of the code also pattern matches against the first list, but this time it covers the `Cons` case. So whenever there is a `Cons` in the first list, as a result, we return this element `Cons x` appended recursively to `add' xs ys`, where `xs` is the remainder of the first list and `ys` is the second list. Example usage:

```
Idris> add' (Cons 1 (Cons 2 (Cons 3 End))) (Cons 4 End)
Cons 1 (Cons 2 (Cons 3 (Cons 4 End))) : MyList Integer
```

Exercise 9

Unfold `add' (Cons 1 (Cons 2 (Cons 3 End))) (Cons 4 End)` on paper to get a better understanding of how this definition appends two lists.

Exercise 10

Come up with a definition for `length'`, which should return the number of elements, given a list.

Hint: The type is `length' : MyList a -> Nat`

Exercise 11

Come up with a definition for `even_members`, which should return a new list with even natural numbers only.

Hint: The type is `even_members : MyList Nat -> MyList Nat`. You can re-use the definition of `even` we've discussed earlier.

Exercise 12

Come up with a definition for `sum'`, which should return a number that will be the sum of all elements in a list of natural numbers.

Hint: The type is `sum' : MyList Nat -> Nat`.

4.1.6. Total and partial functions

Definition 3

A total function is a function that terminates (or returns a value) for all possible inputs.

A partial function is the opposite of a total function. If a function is total, its type can be understood as a precise description of what that function can do. Idris differentiates total from partial functions but allows defining both[28]. As an example, if we assume that we have a function that returns a `String`, then:

1. If it's total, it will return a `String` in finite time
2. If it's partial, then unless it crashes or enters in an infinite loop, it will return a `String`

In Idris, to define partial/total functions we just put the keyword `partial`/`total` in front of the function definition. For example, for the following program we define two functions `test` and `test2`, a partial and a total one respectively:

[28] Partial (non-terminating) functions are what make Idris Turing complete.

```
partial
test : Nat -> String
test Z = "Hi"

total
test2 : Nat -> String
test2 Z = "Hi"
test2 _ = "Hello"
```

If we try to interact with these functions, we get the following results:

```
Idris> test 0
"Hi" : String
Idris> test 1
test 1 : String
Idris> test2 0
"Hi" : String
Idris> test2 1
"Hello" : String
```

We can note that the evaluation of `test 1` does not produce a computed value as a result. Note that at compile-time, Idris will **evaluate the types only for total functions**.

Exercise 13

Try to define a function to be `total`, and at the same time make sure you are not covering all input cases. Note what errors will Idris return in this case.

4.1.7. Higher-order functions

Definition 4

A higher-order function is a function that takes one or more functions as parameters or returns a function as a result.

There are three built-in higher-order functions that are generally useful: `map`, `filter`, `fold` (left and right). Here's the description of each:

1. `map` is a function that takes as input a function with a single parameter and a list and returns a list where all members of the list have this function applied to them
2. `filter` is a function that takes as input a function (predicate) with a single parameter (that returns a `Bool`) and a list and only returns those members in the list whose predicate evaluates to `True`
3. `fold` is a function that takes as input a combining function that accepts two parameters (current value and accumulator), an initial value and a list and returns a value combined with this function. There are two types of folds, a left and a right one, which combines from the left and the right respectively

As an example usage:

```
Idris> map (\x => x + 1) [1, 2, 3]
[2, 3, 4] : List Integer
Idris> filter (\x => x > 1) [1, 2, 3]
[2, 3] : List Integer
Idris> foldl (\x, y => x + y) 0 [1, 2, 3]
6 : Integer
Idris> foldr (\x, y => x + y) 0 [1, 2, 3]
6 : Integer
```

We can actually implement the `map` function ourselves:

```
mymap : (a -> a) -> List a -> List a
mymap _ []        = []
mymap f (x::xs) = (f x) :: (mymap f xs)
```

Note that `::` is used by the built-in `List` type, and is equivalent to `Cons` we've used earlier. However, since `::` is an infix operator, it has to go between the two arguments. The value `[]` represents the empty list and is equivalent to `End`. In addition, the built-in `List` type is polymorphic.

Exercise 14

Do a few different calculations with `mymap` in order to get a deeper understanding of how it works.

Exercise 15

Implement a function `myfilter` that acts just like the `filter` function.

Hint: Use `:t filter` to get its type.

Exercise 16

Given `foldl (\x, y => [y] ++ x) [] [1, 2, 3]` and `foldr (\x, y => y ++ [x]) [] [1, 2, 3]`:

1. Evaluate both of them in Idris to see the values produced.
2. Try to understand the differences between the two expressions.
3. Remove the square brackets `[` and `]` in the lambda body to see what errors Idris produces.
4. Evaluate them on paper to figure out why they produce the given results.

4.1.8. Dependent types

We will implement the `List n` data type that we discussed in section 3.3, which should limit the length of a list at the type level. Idris already has a built-in list like this called `Vect`, so to not conflict we'll name it `MyVect`. We can implement it as follows:

```
data MyVect : (n : Nat) -> Type where
    Empty : MyVect 0
    Cons  : (x : Nat) -> (xs : MyVect len) -> MyVect (S len)
```

We created a new type called `MyVect` which accepts a natural number `n` and returns a `Type`, that is joined with two value constructors:

1. `Empty` - which is just the empty vector (list)
2. `Cons : (x : Nat) -> (xs : MyVect len) -> MyVect (S len)` - which, given a natural number `x` and a vector `xs` of length `len`, will return a vector of length `S len`, that is, `len + 1`.

Note how we additionally specified names to the parameters (`n : Nat`, `xs : MyVect len`, etc.). This can be useful if we want to reference those parameters elsewhere in the type definition.

If we now use the following code snippet, it will pass the compile-time checks:

```
x : MyVect 2
x = Cons 1 (Cons 2 Empty)
```

However, if we try to use this code snippet instead:

```
x : MyVect 3
x = Cons 1 (Cons 2 Empty)
```

we will get the following error:

```
Type mismatch between
    MyVect 0 (Type of Empty)
and
    MyVect 1 (Expected type)
```

Which is a way of Idris telling us that our types do not match and that it cannot verify the “proof” provided.

In this example, we implemented a dependent type that puts the length of a list at the type level. In other programming languages that do not support dependent types, this is usually checked at the code level (run-time) and compile-time checks are not able to verify this.

One example where such a guarantee might be useful is in preventing buffer overflows. We could encode the dimension of an array at the type level, and statically guarantee that array reads and writes only happen within bounds.

Exercise 17

Come up with a function `isSingleton` that accepts a `Bool` and returns a `Type`. This function should return an object of type `Nat` in the `True` case, and `MyVect Nat` otherwise. Further, implement a function `mkSingle` that accepts a `Bool`, and returns `isSingleton True` or `isSingleton False`, and as a computed value will either return `0` or `Empty`.

Hint: The data definitions are `isSingleton : Bool -> Type` and `mkSingle : (x : Bool) -> isSingleton x` respectively.

4.1.9. Implicit parameters

Implicit parameters (arguments) allow us to bring values from the type level to the program level. At the program level, by using curly braces we allow them to be used in the definition of the function. Let’s take a look at the following example, which uses our dependent type `MyVect` that we defined earlier:

```
lengthMyVect : MyVect n -> Nat
lengthMyVect {n = k} list = k
```

In this case, we defined a function `lengthMyVect` that takes a `MyVect` and returns a natural number. The value `n` in the definition of the function will be the same as the value of `n` at the type level. They are called implicit parameters because the caller of this function needn’t pass these parameters. In the function definition, we define implicit parameters with curly braces and we also need to specify the list parameter which is of type `MyVect n` to pattern match against it. But, note how we don’t refer to the list parameter in the computation part of this function and instead, we can use an underscore (which represents an unused parameter) to get to:

```
lengthMyVect : MyVect n -> Nat
lengthMyVect {n = k} _ = k
```

We can also have implicit parameters at the type level. As a matter of fact, an equivalent type definition of that function is:

```
lengthMyVect : {n : Nat} -> MyVect n -> Nat
```

If we ask Idris to give us the type of this function, we will get the following for either of the type definitions above:

```
Idris> :t lengthMyVect
lengthMyVect : MyVect n -> Nat
```

However, we can use the command `:set showimplicits` which will show the implicits on the type level. If we do that, we will get the following for either of the type definitions above:

```
Idris> :set showimplicits
Idris> :t lengthMyVect
lengthMyVect : {n : Nat} -> MyVect n -> Nat
```

To pass values for implicit arguments, we can use the following syntax:

```
Idris> lengthMyVect {n = 1} (Cons 1 Empty)
1 : Nat
```

Exercise 18

Try to evaluate the following code and observe the results:

```
lengthMyVect {n = 2} (Cons 1 Empty)
```

4.1.10. Pattern matching expressions

We've seen how pattern matching is a powerful concept, in that it allows us to pattern match against value constructors. For example, we can write a filtering function that given a list of naturals produces a list of even naturals, by re-using the earlier definition of `even`:

```
total even_members : MyList Nat -> MyList Nat
even_members End          = End
even_members (Cons x l') = if (even x)
                            then (Cons x (even_members l'))
                            else even_members l'
```

The function above is a recursive one, and depending on the value of `even x` it will branch the recursion. Since pattern matching works against value constructors, and `even x` is a function call, we can't easily pattern match against it. We used `even x` in the function body to do the check. Idris provides another additional keyword `with` that allows us to pattern match a value of some expression. The keyword `with` has the following syntax:

```
function (pattern_match_1) with (expression)
    pattern_match_1' | (value of expression to match) = ...
    pattern_match_2' | (value of expression to match) = ...
    ...
```

Note how we have to specify new pattern matching clauses after the line that uses the `with` keyword. This is so because we won't have the original pattern match in context. Given this, an alternative definition of the function above is:

```
total even_members' : MyList Nat -> MyList Nat
even_members' End = End
even_members' (Cons x l') with (even x)
  even_members' (Cons x l') | True  = Cons x (even_members' l')
  even_members' (Cons _ l') | False = (even_members' l')
```

In this function, we defined two new pattern matches after the line that uses the `with` keyword. Since we don't have `x` and `l'` in this new pattern matching context, we have to rewrite them on the left side of the pipe, and on the right side of the pipe we pattern match against the value of `even x`, and then branch the recursion (computation).

4.1.11. Interfaces and implementations

Interfaces are defined using the `interface` keyword and they allow us to add constraints to types that implement them[29]. As an example, we'll take a look at the `Eq` interface:

```
interface Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool
        -- Minimal complete definition:
        --      (==) or (/=)
    x /= y = not (x == y)
    x == y = not (x /= y)
```

Note how we can specify comments in the code by using two dashes. Comments are ignored by the Idris compiler and are only useful to the reader of the code.

The definition says that for a type to implement the `Eq` interface, there must be an implementation of the functions == and /= for that specific type. Additionally, the interface also contains definitions for the functions, but this is optional. Since the definition of == depends on /= (and vice-versa), it will be sufficient to provide only one of them in the implementation, and the other one will be automatically generated.

As an example, let's assume that we have a data type:

[29] They are equivalent to Haskell's `class` keyword. Interfaces in Idris are very similar to OOP's interfaces.

```
data Foo : Type where
    Fooinst : Nat -> String -> Foo
```

To implement Eq for Foo, we can use the following code:

```
implementation Eq Foo where
    (Fooinst x1 str1) ==
        (Fooinst x2 str2) = (x1 == x2) && (str1 == str2)
```

We use == for Nat and String since this is already defined in Idris itself. With this, we can easily use == and /= on Fooinst:

```
Idris> Fooinst 3 "orange" == Fooinst 6 "apple"
False : Bool
Idris> Fooinst 3 "orange" /= Fooinst 6 "apple"
True : Bool
```

Nats implement the built-in Num interface, which is what allows us to use 0 and Z interchangeably.

Exercise 19

Implement your own data type Person that accepts a person's name and age, and implement an interface for comparing Persons.

Hint: One valid data type is:

```
data Person = Personinst String Int
```

4.2. Curry-Howard isomorphism

The Curry-Howard isomorphism (also known as Curry-Howard correspondence) is the direct relationship between computer programs and mathematical proofs. It is named after the mathematician Haskell Curry and logician William Howard. In other words, a mathematical proof is represented by a computer program and the formula that we're proving is the type of that program. As an example, we can take a look at the function `swap` that is defined as follows:

```
swap : (a, b) -> (b, a)
swap (a, b) = (b, a)
```

The isomorphism says that this function has an equivalent form of mathematical proof. Although it may not be immediately obvious, let's consider the following proof: Given $P \wedge Q$, prove that $Q \wedge P$. In order to prove it, we have to use the inference rules and-introduction and and-elimination, which are defined as follows:

1. And-introduction means that if we are given P, Q, then we can construct a proof for $P \wedge Q$
2. Left and-elimination means that if we are given $P \wedge Q$, we can conclude P
3. Right and-elimination means that if we are given $P \wedge Q$, we can conclude Q

If we want to implement this proof in Idris, we can represent it as a product type as follows:

```
data And a b = And_intro a b

and_comm : And a b -> And b a
and_comm (And_intro a b) = And_intro b a
```

As we've discussed, we can use product types to encode pairs. We can note the following similarities with our earlier definition of `swap`:

1. `And_intro x y` is equivalent to constructing a product type `(x, y)`
2. Left-elimination, which is a pattern match of `And_intro a _` is equivalent to the first element of the product type
3. Right-elimination, which is a pattern match of `And_intro _ b` is equivalent to the second element of the product type

As long as Idris' type checker terminates, we can be certain that the program provides a mathematical proof of its type. This is why Idris' type checker only evaluates total functions, to keep the type checking decidable.

4.3. Quantitative Type Theory

This book was written using Idris 1, however, in 2020 a new version, Idris 2 was released which is based on **Quantitative Type Theory**. Most of the examples in this book should be compatible with Idris 2. However, some Idris 2 code will not be compatible with Idris 1. Idris 2 is recommended as it also might contain some bug fixes.

Quantitative type theory gives more computational power at the type level. It allows us to specify a quantity for each variable:

- 0 - which means that the variable is not used at run-time
- 1 - which means that the variable is used only once at run-time
- Unrestricted (default) - which is the same behavior as Idris 1

Consider the `lengthMyVect` example from before. If we change its definition from `k` to `k + k`, the type checker will not complain:

```
lengthMyVect : {n : Nat} -> MyVect n -> Nat
lengthMyVect {n = k} _ = k + k
```

However, if we specify that the variable `n` can be used only once:

```
lengthMyVect : {1 n : Nat} -> MyVect n -> Nat
```

Then the type checker will throw an error for `k + k`, but not for `k`.

5. Proving in Idris

In this chapter, we will provide several examples to demonstrate the power of Idris. We will do mathematical proofs. There are a lot of Idris built-ins that will help us achieve our goals and in each section, we will introduce the relevant definitions.

Definition 1

The equality data type is roughly defined as follows:

```
data (=) : a -> b -> Type where
    Refl : x = x
```

We can use the value constructor `Refl` to prove equalities.

Definition 2

The function `the` accepts `A : Type` and `value : A` and then returns `value : A`. We can use it to manually assign a type to an expression.

Exercise 1

Check the documentation of the equality type with `:doc (=)`.

Exercise 2

Evaluate `the Nat 3` and `the Integer 3` and note the differences. Afterward, try to implement `the'` that will act just like `the` and test the previous evaluations again.

Exercise 3

Given `data Or a b = Or_introl a | Or_intror b`, show that $a \to (a \lor b)$ and $b \to (a \lor b)$. Afterwards, check the documentation of the built-in `Either` and compare it to our `Or`.

Hint:

```
proof_1 : a -> Or a b
proof_1 a = Or_introl ?prf
```

Exercise 4

In section 4.2 we implemented `And`. How does it compare to the built-in `Pair`?

5.1. Weekdays

In this section, we will introduce a way to represent weekdays and then do some proofs with them. We start with the following data structure:

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

The `Weekday` type has 7 value constructors, one for each weekday. The data type and its constructors do not accept any parameters.

5.1.1. First proof (auto-inference)

We will prove that after Monday comes Tuesday. We start by implementing a function that, given a weekday, returns the next weekday:

```
total next_day : Weekday -> Weekday
next_day Mon = Tue
next_day Tue = Wed
next_day Wed = Thu
next_day Thu = Fri
next_day Fri = Sat
next_day Sat = Sun
next_day Sun = Mon
```

Given these definitions, we can write the proof as follows:

```
our_first_proof : next_day Mon = Tue
our_first_proof = ?prf
```

Note how we used the function `next_day Mon = Tue` at the type level, for the type of `our_first_proof`. Since `next_day` is total, Idris will be able to evaluate `next_day Mon` at compile-time. We used a hole for the definition since we still don't know what the proof will look like. If we run this code in Idris, it will tell us we have a hole `prf` to fill:

```
Type checking ./first_proof.idr
Holes: Main.prf
Idris> :t prf
--------------------------------------
prf : Tue = Tue
```

Checking the type of `prf` we notice how Idris evaluated the left part of the equation at compile-time. In order to prove that `Tue = Tue`, we can just use `Refl`:

```
our_first_proof : next_day Mon = Tue
our_first_proof = Refl
```

Reloading the file in Idris:

```
Idris> :r
Type checking ./first_proof.idr
```

The type check was successful. Per the Curry-Howard isomorphism, this means that we've successfully proven that `next_day Mon = Tue`. So, after Monday comes Tuesday!

Exercise 5

Remove one or more pattern match definitions of `next_day` and observe the error that Idris will produce. Afterward, alter the function so that it is not total anymore, and observe the results.

Exercise 6

Implement `prev_day` and prove that Sunday is before Monday.

5.1.2. Second proof (rewrite)

In addition to the previous proof, we'll implement a function that accepts a `Weekday` and returns `True` if it's Monday and `False` otherwise.

```
is_it_monday : Weekday -> Bool
is_it_monday Mon = True
is_it_monday _   = False
```

For the sake of example, we will prove that for any given day, if it's Monday then `is_it_monday` will return `True`. It's obvious from the definition of `is_it_monday`, but proving that is a whole different story. The type definition that we need to prove is:

```
our_second_proof : (day : Weekday) -> day = Mon ->
    is_it_monday day = True
```

We gave a name to the first parameter `day : Weekday` so that we can refer to it in the rest of the type definition. The second parameter says that `day = Mon` and the return value is `is_it_monday day = True`. We can treat the first and the second parameter as givens since we are allowed to assume them (per definition of implication). With that, we proceed to the function definition:

```
our_second_proof day day_eq_Mon = Refl
```

In this definition, `day` and `day_eq_Mon` are our assumptions (givens). If we run this code in Idris, it will produce an error at compile-time since it cannot deduce that `True` is equal to `is_it_monday day`. In the previous proof example, Idris was able to infer everything from the definitions at compile-time. However, at this point, we need to help Idris do the inference since it cannot derive the proof based only on the definitions. We can change the `Refl` to a hole `prf`:

```
  day : Weekday
  day_eq_Mon : day = Mon
--------------------------------------
prf : is_it_monday day = True
```

Note how checking the type of the hole lists the givens/premises (above the separator), and our goal(s) (below the separator). We see that along with `prf` we also get `day` and `day_eq_Mon` in the list of givens, per the left-hand side of the function definition of `our_second_proof`.

How do we replace something we have in the givens, with the goal?

If we only had a way to tell Idris that it just needs to replace `day` with `day = Mon` to get to `is_it_monday Mon = True`, it will be able to infer the rest.

Definition 3

The `rewrite` keyword can be used to rewrite expressions. If we have `X : x = y`, then the syntax `rewrite X in Y` will replace all occurrences of `x` with `y` in `Y`.

With the power to do rewrites, we can attempt the proof as follows:

```
our_second_proof : (day : Weekday) -> day = Mon ->
    is_it_monday day = True
our_second_proof day day_eq_Mon = rewrite day_eq_Mon in ?prf
```

Idris produces:

```
Idris> :t prf
  day : Weekday
  day_eq_Mon : day = Mon
  _rewrite_rule : day = Mon
--------------------------------------
prf : True = True
```

Changing `prf` to `Refl` completes the proof. We just proved that $\forall x \in \text{Weekdays}, x = \text{Mon} \to IsItMonday(x)$. We assumed $x = \text{Mon}$ is true (by pattern matching against `day_eq_Mon` in our definition), and then used rewriting to alter `x`.

Exercise 7

Implement the function `is_it_sunday` that returns `True` if the given day is Sunday, and `False` otherwise.

Exercise 8

Prove the following formula in Idris: $\forall x \in \text{Weekdays}, x = \text{Sun} \to IsItSunday(x)$.

5.1.3. Third proof (impossible)

In this section, we will prove that `is_it_monday Tue = True` is a contradiction.

Per intuitionistic logic, in order to prove that P is a contradiction, we need to prove $P \to \bot$. Idris provides an empty data type `Void`. This data type has no value constructors (proofs) for it.

To prove that `is_it_monday Tue = True` is a contradiction, we will do the following:

```
our_third_proof : is_it_monday Tue = True -> Void
our_third_proof mon_is_Tue = ?prf
```

Checking the type of the hole:

```
  mon_is_Tue : False = True
--------------------------------------
prf : Void
```

How do we prove `prf`, given that there are no value constructors for `Void`?

Seems that at this point we are stuck. We need to find a way to tell Idris that this proof is impossible.

Definition 4

The `impossible` keyword can be used to prove statements that are not true. With this keyword, we say that proof for a data type cannot be constructed since there does not exist a value constructor for that particular type.

We will slightly rewrite the function:

```
our_third_proof : is_it_monday Tue = True -> Void
our_third_proof Refl impossible
```

With this syntax, we're telling Idris that the reflexivity of `False = True` is impossible and thus the proof is complete.

Exercise 9

Check the documentation of `Void` and try to implement `Void'` yourself. Rewrite the proof above to use `Void'` instead of `Void`.

Exercise 10

Prove that `1 = 2` is a contradiction.

Hint: The type is `1 = 2 -> Void`

5.2. Natural numbers

In this section, we will prove facts about natural numbers and also do some induction. Recall that a natural number is defined either as zero or as the successor of a natural number. So, `0, S 0, S (S 0), ...` are the first natural numbers. We will start with the following definitions:

```
data MyNat = Zero | Succ MyNat

total mynat_plus : MyNat -> MyNat -> MyNat
mynat_plus Zero     m = m
mynat_plus (Succ k) m = Succ (mynat_plus k m)
```

Note how the definition of `MyNat` is recursive compared to `Weekday`. A consequence of that is that we may need to use induction for some proofs.

Exercise 11

Compare the addition definition to Definition 21 in chapter 2.

5.2.1. First proof (auto-inference and existence)

We will prove that $0 + a = a$, given the definitions for natural numbers and addition.

For that, we need to implement a function that accepts a natural number `a` and returns the proposition that `mynat_plus Zero a = a`.

```
total our_first_proof_inf : (a : MyNat) -> mynat_plus Zero a = a
our_first_proof_inf a = ?prf
```

If we check the type of the hole, we get that the goal is `prf : a = a`, so changing the hole to a `Refl` completes the proof. Idris was able to automatically infer the proof by directly substituting definitions.

To prove the existence of a successor, i.e. `Succ x`, per intuitionistic logic we need to construct a pair where the first element is `x : MyNat` and the second element is `Succ x : MyNat`. Idris has a built-in data structure for constructing dependent pairs called `DPair`.

```
total our_first_proof_exist : MyNat -> DPair MyNat (\_ => MyNat)
our_first_proof_exist x = MkDPair x (Succ x)
```

We just proved that $\exists x \in \text{MyNat}, Succ(x)$.

Exercise 12

Check the documentation of `DPair` and `MkDPair` and try to construct some dependent pairs.

5.2.2. Second proof (introduction of a new given)

An alternative way to prove that a natural number exists is as follows:

```
total our_second_proof : MyNat
our_second_proof = ?prf
```

If we check the type of the hole, we get:

```
Holes: Main.prf
Idris> :t prf
--------------------------------------
prf : MyNat
```

By the definition of `MyNat` we are sure that there exists a value constructor of `MyNat`, but how do we tell Idris?

Definition 5

The `let` keyword that we introduced earlier allows us to add a new given to the list of hypotheses.

We can slightly rewrite our code:

```
total our_second_proof : MyNat
our_second_proof = let the_number = Zero in ?prf
```

Checking the type:

```
  the_number : MyNat
--------------------------------------
prf : MyNat
```

Changing `prf` to `the_number` concludes the proof.

Exercise 13

Simplify `our_second_proof` without the use of `let`.

Hint: Providing a valid value constructor that satisfies (inhabits) the type is a constructive proof.

Exercise 14

Construct a proof similar to `our_second_proof` without defining a function for it and by using the function `the`.

5.2.3. Third proof (induction)

We will prove that $a + 0 = a$. We can try to use the same approach as in 5.2.1:

```
total our_third_proof : (a : MyNat) -> mynat_plus a Zero = a
our_third_proof a = Refl
```

If we try to run the code above, Idris will produce an error saying that there is a type mismatch between `a` and `mynat_plus a Zero`.

It seems that we have just about all the definitions we need, but we're missing a piece. How do we re-use our definitions?

To prove that $a + 0 = a$, we can use mathematical induction starting with the definitions we already have as the base case and build on top of that until $a + 0 = a$. That is, we need to prove that $0 + 0 = 0 \to (a - a) + 0 = a - a \to ... \to (a - 1) + 0 = a - 1 \to a + 0 = a$.

From here, we rewrite our function to contain a base case and an inductive step:

```
total our_third_proof : (a : MyNat) -> mynat_plus a Zero = a
our_third_proof Zero     = ?base
our_third_proof (Succ k) = ?ind_hypothesis
```

Note how we used pattern matching against the definition of natural numbers. Pattern matching is similar to using proof by cases. Checking the types of the holes:

```
Holes: Main.ind_hypothesis, Main.base
Idris> :t base
--------------------------------------
base : Zero = Zero
Idris> :t ind_hypothesis
  k : MyNat
--------------------------------------
ind_hypothesis : Succ (mynat_plus k Zero) = Succ k
```

For the base case we can just use `Refl`, but for the inductive step we need to do something different. We need to find a way to assume (add to list of givens) $a+0=a$ and show that $(a+1)+0=a+1$ follows from that assumption. Since we pattern match on `Succ k`, we can use recursion on `k` along with `let` to generate the hypothesis:

```
total our_third_proof : (a : MyNat) -> mynat_plus a Zero = a
our_third_proof Zero     = Refl
our_third_proof (Succ k) = let ind_hypothesis = our_third_proof k in
                           ?conclusion
```

Our proof givens and goals become:

```
  k : MyNat
  ind_hypothesis : mynat_plus k Zero = k
--------------------------------------
conclusion : Succ (mynat_plus k Zero) = Succ k
```

To prove the conclusion, we can simply rewrite the inductive hypothesis in the goal and we are done.

```
total our_third_proof : (a : MyNat) -> mynat_plus a Zero = a
our_third_proof Zero     = Refl
our_third_proof (Succ k) = let ind_hypothesis = our_third_proof k in
                           rewrite ind_hypothesis in
                           Refl
```

This concludes the proof.

Exercise 15

Observe the similarity between this proof and the proof in section 2.3.4.

5.2.4. Ordering

Idris 1 has a built-in data type for the ordering of natural numbers `LTE`, which stands for less than or equal to. This data type has two constructors:

1. `LTEZero`, used to prove that zero is less than or equal to any natural number
2. `LTESucc`, used to prove that $a \leq b \to S(a) \leq S(b)$

If we check the type of `LTEZero`, we will get the following:

```
Idris> :t LTEZero
LTEZero : LTE 0 right
```

`LTEZero` does not accept any arguments, but we can pass `right` at the type level. With the use of implicits, we can construct a very simple proof to show that $0 \leq 1$:

```
Idris> LTEZero {right = S Z}
LTEZero : LTE 0 1
```

Similarly, with `LTESucc` we can do the same:

```
Idris> LTESucc {left = Z} {right = S Z}
LTESucc : LTE 0 1 -> LTE 1 2
```

Exercise 16

Check the documentation of `GTE`, and then evaluate `GTE 2 2`. Observe what Idris returns and think about how `GTE` can be implemented in terms of `LTE`.

Exercise 17

We used the built-in type `LTE` which is defined for `Nat`. Try to come up with a `LTE` definition for `MyNat`.

5.2.5. Safe division

Idris 1 provides a function called `divNat` that divides two numbers. Checking the documentation:

```
Idris> :doc divNat
Prelude.Nat.divNat : Nat -> Nat -> Nat

    The function is not total as there are missing cases
```

We can try to use it a couple of times:

```
Idris> divNat 4 2
2 : Nat
Idris> divNat 4 1
4 : Nat
Idris> divNat 4 0
divNat 4 0 : Nat
```

As expected, partial functions do not cover all inputs and thus `divNat` does not return a computed value when we divide by zero.

How do we make a function like `divNat` total?

The only way to make this work is to pass a proof to `divNat` that says that the divisor is not zero. Idris has a built-in function for that, called `divNatNZ`.

We can check the documentation of this function as follows:

```
Idris> :doc divNatNZ
Prelude.Nat.divNatNZ : Nat -> (y : Nat) -> Not (y = 0) -> Nat
    Division where the divisor is not zero.
    The function is Total
```

This function is total, but we need to also provide a parameter (proof) that the divisor is not zero. Fortunately, Idris 1 also provides a function called `SIsNotZ`, which accepts any natural number (through implicit argument `x`) and returns a proof that `x + 1` is not zero.

We can try to construct a few proofs:

```
Idris> SIsNotZ {x = 0}
SIsNotZ : (1 = 0) -> Void
Idris> SIsNotZ {x = 1}
SIsNotZ : (2 = 0) -> Void
Idris> SIsNotZ {x = 2}
SIsNotZ : (3 = 0) -> Void
```

Great. It seems we have everything we need. We can safely divide as follows:

```
Idris> divNatNZ 4 2 (SIsNotZ {x = 1})
2 : Nat
Idris> divNatNZ 4 1 (SIsNotZ {x = 0})
4 : Nat
Idris> divNatNZ 4 0 (SIsNotZ {x = ???})
4 : Nat
```

We cannot construct a proof for the third case and so it will never be able to divide by zero, which is not allowed anyway.

Exercise 18

Implement `SuccIsNotZ` for `MyNat` that works similarly to `SIsNotZ`.

Exercise 19

Implement `minus` and `lte` (and optionally `div`) for `MyNat`.

5.2.6. Maximum of two numbers

Definition 6

The maximum of two numbers a and b is defined as:

$$max(a, b) = \begin{cases} b, \text{ if } a \leq b \\ a, \text{ otherwise} \end{cases}$$

In this section we will try to prove that $b \leq a \to b = max(a, b)$. Idris 1 already has a built-in function `maximum`, so we can re-use that. Next, we need to figure out the type of the function to approach the proof. Intuitively, we can try the following:

```
our_proof : (a : Nat) -> (b : Nat) -> a <= b -> maximum a b = b
our_proof a b a_lt_b = ?prf
```

However this won't work since `a <= b` is a `Bool` (per the function `<=`), not a `Type`. At the type level, we need to rely on `LTE` which is a `Type`.

```
our_proof : (a : Nat) -> (b : Nat) -> LTE a b -> maximum a b = b
our_proof a b a_lt_b = ?prf
```

This compiles and we have to figure out the hole. If we check its type, we get:

```
  a : Nat
  b : Nat
  a_lt_b : LTE a b
--------------------------------------
prf : maximum a b = b
```

This looks a bit complicated, so we can further simplify by breaking the proof into several cases by adding pattern matching for all combinations of the parameters' value constructors:

```
our_proof : (a : Nat) -> (b : Nat) -> LTE a b -> maximum a b = b
our_proof Z Z _              = Refl
our_proof Z (S k) _          = Refl
our_proof (S k) (S j) a_lt_b = ?prf
```

We get the following:

```
  k : Nat
  j : Nat
  a_lt_b : LTE (S k) (S j)
--------------------------------------
prf : S (maximum k j) = S j
```

It seems like we made progress, as this gives us something to work with. We can use induction on `k` and `j`, and use a hole for the third parameter to ask Idris what type we need to satisfy:

```
our_proof : (a : Nat) -> (b : Nat) -> LTE a b -> maximum a b = b
our_proof Z Z _              = Refl
our_proof Z (S k) _          = Refl
our_proof (S k) (S j) a_lt_b = let I_H = (our_proof k j ?prf) in
                                 rewrite IH in
                                 Refl
```

The hole produces the following:

```
Holes: Main.prf
Idris> :t prf
  k : Nat
  j : Nat
  a_lt_b : LTE (S k) (S j)
--------------------------------------
prf : LTE k j
```

How do we go from $S(a) \leq S(b) \to a \leq b$?

It seems pretty obvious that if we know that $1 \leq 2$, then also $0 \leq 1$, but we still need to find out how to tell Idris that this is true. For this, Idris has a built-in function `fromLteSucc`:

```
Idris> :doc fromLteSucc
Prelude.Nat.fromLteSucc : LTE (S m) (S n) -> LTE m n
    If two numbers are ordered, their predecessors are too
```

It seems we have everything we need to conclude our proof. We can proceed as follows:

```
total
our_proof : (a : Nat) -> (b : Nat) -> LTE a b -> maximum a b = b
our_proof Z Z _              = Refl
our_proof Z (S k) _          = Refl
our_proof (S k) (S j) a_lt_b = let fls = fromLteSucc a_lt_b in
                               let IH  = (our_proof k j fls) in
                               rewrite IH in
                               Refl
```

or a more simplified version:

```
total
our_proof : (a : Nat) -> (b : Nat) -> LTE a b -> maximum a b = b
our_proof Z Z _              = Refl
our_proof Z (S k) _          = Refl
our_proof (S k) (S j) a_lt_b = rewrite
                               (our_proof k j (fromLteSucc a_lt_b)) in
                               Refl
```

Exercise 20

Use `fromLteSucc` with implicits to construct some proofs.

5.2.7. List of even naturals

We will prove that a list of even numbers contains no odd numbers. We will re-use the functions `even` in 4.1.4 and `even_members` in 4.1.11. We will also need another function to check if a list has odd numbers:

```
total has_odd : MyList Nat -> Bool
has_odd End          = False
has_odd (Cons x l') = if (even x) then has_odd l' else True
```

To prove that a list of even numbers contains no odd numbers, we can use the following type definition:

```
even_members_list_only_even : (l : MyList Nat) ->
    has_odd (even_members l) = False
```

Note that `has_odd` is branching computation depending on the value of `even x`, so we have to pattern match with the value of expressions by using the keyword `with`. The base case is simply `Refl`:

```
even_members_list_only_even End = Refl
```

However, for the inductive step, we will use `with` on `even n` and produce proof depending on the evenness of the number:

```
even_members_list_only_even (Cons n l') with (even n) proof even_n
  even_members_list_only_even (Cons n l') | False =
      let IH = even_members_list_only_even l' in ?a
  even_members_list_only_even (Cons n l') | True  =
      let IH = even_members_list_only_even l' in ?b
```

Note how we specified `proof even_n` right after the expression in the `with` match. The `proof` keyword followed by a variable brings us the proof of the expression to the list of premises. The expression `with (even n) proof even_n` will pattern match on the results of `even n`, and will also bring the proof `even n` in the premises. If we now check the first hole:

```
  n : Nat
  l' : MyList Nat
  even_n : False = even n
  IH : has_odd (even_members l') = False
--------------------------------------
a : has_odd (even_members l') = False
```

That should be simple, we can just use `IH` to solve the goal. For the second hole, we have:

```
  n : Nat
  l' : MyList Nat
  even_n : True = even n
  IH : has_odd (even_members l') = False
--------------------------------------
b : ifThenElse (even n) (has_odd (even_members l'))
    True = False
```

How do we rewrite the inductive hypothesis to the goal in this case?

It seems that we can't just rewrite here since `even_n` has the order of the equality reversed. Idris provides a function called `sym` which takes an equality of `a = b` and converts it to `b = a`.

We can try to rewrite `sym even_n` to the goal, and it now becomes:

```
  n : Nat
  l' : MyList Nat
  even_n : True = even n
  IH : has_odd (even_members l') = False
  _rewrite_rule : even n = True
--------------------------------------
b : has_odd (even_members l') = False
```

Similarly to before we will use `IH` to solve the goal. Thus, the complete proof:

```
even_members_list_only_even : (l : MyList Nat) ->
    has_odd (even_members l) = False
even_members_list_only_even End = Refl
even_members_list_only_even (Cons n l') with (even n) proof even_n
  even_members_list_only_even (Cons n l') | False =
      let IH = even_members_list_only_even l' in IH
  even_members_list_only_even (Cons n l') | True  =
      let IH = even_members_list_only_even l' in
      rewrite sym even_n in IH
```

How did mathematical induction work in this case?

Mathematical induction is defined in terms of natural numbers, but in this case, we used induction to prove a fact about a list. This works because we used a more general induction called *structural induction*. Structural induction is used to prove that some proposition $P(x)$ holds for all x of some sort of recursively defined data structure. For example, for lists, we used `End` as the base case and `Cons` as the inductive step. Thus, mathematical induction is a special case of structural induction for the `Nat` type.

Exercise 21

Rewrite `has_odd` to use `with` in the recursive case, and then repeat the proof above.

5.2.8. Partial orders

Definition 7

A binary relation R on some set S is a partial order if the following properties are satisfied:

1. $\forall a \in S, aRa$, i.e. reflexivity
2. $\forall a, b, c \in S, aRb \land bRc \to aRc$, i.e. transitivity
3. $\forall a, b \in S, aRb \land bRa \to a = b$, i.e. antisymmetry

Let's abstract this in Idris as an `interface`:

```
interface Porder (a : Type) (Order : a -> a -> Type) | Order where
    total proofR : Order n n -- reflexivity
    total proofT : Order n m -> Order m p -> Order n p -- transitivity
    total proofA : Order n m -> Order m n -> n = m -- antisymmetry
```

The interface `Porder` accepts a `Type` and a relation `Order` which is a built-in binary function in Idris 1. Since the interface has more than two parameters, we specify that `Order` is a determining parameter, i.e. the parameter used to resolve the instance.

Now that we have our abstract interface we can build a concrete implementation for it:

```
implementation Porder Nat LTE where
    proofR {n = Z}   =
        LTEZero
    proofR {n = S _} =
        LTESucc proofR

    proofT LTEZero           _                 =
        LTEZero
    proofT (LTESucc n_lte_m) (LTESucc m_lte_p) =
        LTESucc (proofT n_lte_m m_lte_p)

    proofA LTEZero           LTEZero           =
        Refl
    proofA (LTESucc n_lte_m) (LTESucc m_lte_n) =
        let IH = proofA n_lte_m m_lte_n in rewrite IH in Refl
```

We proved that the binary operation "less than or equal to" for `Nat`s make a `Porder`. Interfaces allow us to group one or more functions, and implementation of a specific interface is guaranteed to implement all such functions.

Exercise 22

Convince yourself using pen and paper that ≤ on natural numbers makes a partial order, i.e. it satisfies all properties of Definition 7. Afterward, try to understand the proofs for reflexivity, transitivity, and antisymmetry by deducing them yourself in Idris using holes.

5.3. Computations as types

As we stated earlier, types are first-class citizens in Idris. In this section, we will see how we can represent computation at the type level.

5.3.1. Same elements in a list (vector)

Re-using the same definition of `MyVect`, we can write a function to test if all elements are the same in a given list:

```
allSame : (xs : MyVect n) -> Bool
allSame Empty                  = True
allSame (Cons x Empty)         = True
allSame (Cons x (Cons y xs)) = x == y && allSame xs
```

Idris will return `True` in case all elements are equal to each other, and `False` otherwise. Let's now think about how we can represent this function in terms of types. We want to have a type `AllSame` that has three constructors:

1. `AllSameZero` which is a proof for `AllSame` in case of an empty list
2. `AllSameOne` which is a proof for `AllSame` in case of a single-element list
3. `AllSameMany` which is a proof for `AllSame` in case of a list with multiple elements

This is how the data type could look like:

```
data AllSame : MyVect n -> Type where
    AllSameZero : AllSame Empty
    AllSameOne  : (x : Nat) -> AllSame (Cons x Empty)
    AllSameMany : (x : Nat) -> (y : Nat) -> (ys : MyVect _) ->
        True = (x == y) -> AllSame (Cons y ys) ->
        AllSame (Cons x (Cons y ys))
```

The constructors `AllSameZero` and `AllSameOne` are easy. However, the recursive constructor `AllSameMany` is a bit trickier. It accepts two natural numbers `x` and `y`, a list `ys`, a proof that `x` and `y` are the same, and a proof that `y` concatenated to `ys` is a same-element list. Given this, it will produce a proof that `x` concatenated to `y` concatenated to `ys` is also a same-element list. This type definition captures exactly the definition of a list that would contain all the same elements.

Interacting with the constructors:

```
Idris> AllSameZero
AllSameZero : AllSame Empty
Idris> AllSameOne 1
AllSameOne 1 : AllSame (Cons 1 Empty)
Idris> AllSameMany 1 1 Empty Refl (AllSameOne 1)
AllSameMany 1 1 Empty Refl (AllSameOne 1) : AllSame (Cons 1
    (Cons 1 Empty))
```

The third example is a proof that the list `[1, 1]` has the same elements. However, if we try to use the constructor with different elements:

```
Idris> AllSameMany 1 2 Empty
AllSameMany 1 2 Empty : (True = False) -> AllSame (Cons 2 Empty) ->
    AllSame (Cons 1 (Cons 2 Empty))
```

We see that Idris requires us to provide proof that `True = False`, which is impossible. So for some lists, the type `AllSame` cannot be constructed, but for some, it can. If we now want to make a function that given a list, it maybe produces a type `AllSame`, we need to consider the `Maybe` data type first which has the following definition:

```
data Maybe a = Just a | Nothing
```

Interacting with it:

```
Idris> the (Maybe Nat) (Just 3)
Just 3 : Maybe Nat
Idris> the (Maybe Nat) Nothing
Nothing : Maybe Nat
```

We can now proceed to write our function:

```
mkAllSame : (xs : MyVect n) -> Maybe (AllSame xs)
mkAllSame Empty                = Just AllSameZero
mkAllSame (Cons x Empty)       = Just (AllSameOne x)
mkAllSame (Cons x (Cons y xs)) with (x == y) proof x_eq_y
    mkAllSame (Cons x (Cons y xs)) | False =
        Nothing
    mkAllSame (Cons x (Cons y xs)) | True  =
        case (mkAllSame (Cons y xs)) of
            Just y_eq_xs => Just (AllSameMany x y xs x_eq_y y_eq_xs)
            Nothing      => Nothing
```

Interacting with it:

```
Idris> mkAllSame (Cons 1 Empty)
Just (AllSameOne 1) : Maybe (AllSame (Cons 1 Empty))
Idris> mkAllSame (Cons 1 (Cons 1 Empty))
Just (AllSameMany 1 1 Empty Refl (AllSameOne 1)) : Maybe (AllSame
    (Cons 1 (Cons 1 Empty)))
Idris> mkAllSame (Cons 1 (Cons 2 Empty))
Nothing : Maybe (AllSame (Cons 1 (Cons 2 Empty)))
```

For lists that contain the same elements it will use the `Just` constructor, and `Nothing` otherwise. Finally, we can rewrite our original `allSame` as follows:

```
allSame' : MyVect n -> Bool
allSame' xs = case (mkAllSame xs) of
    Nothing => False
    Just _  => True
```

5.3.2. Evenness of numbers

Next, we will represent a data type using a natural deduction style and then do the same corresponding definitions in Idris.

Definition 8

In natural deduction style, propositions are represented with a line in the middle where everything above the line are the premises and everything below it is the conclusion.

When there is an implication ($\to$) within one of the rules in natural deduction style, then this implication is thought to be at the object level, while the actual line represents implication at the metalanguage level. We'll see an example of this in Appendix A.

As an example, let the first inference rule be $\frac{}{Ev\ 0}$. Note that there is nothing above the line, so we can think of this as an "axiom". Using the previous notation, this would just be $Ev\ 0$. Further, let the second inference rule be $\frac{Ev\ n}{Ev\ (S\ (S\ n))}$. Using the previous notation, this would be $\forall n, Ev\ n \to Ev\ (S\ (S\ n))$.

The same representation in Idris:

```
data Ev : Nat -> Type where
    EvZero : Ev Z
    EvSucc : (n : Nat) -> Ev n -> Ev (S (S n))
```

Having this definition, we can now construct proofs of even numbers as follows:

```
Idris> EvSucc 0 EvZero
EvSucc 0 EvZero : Ev 2
Idris> EvSucc 2 (EvSucc 0 EvZero)
EvSucc 2 (EvSucc 0 EvZero) : Ev 4
```

We just represented even numbers at the type level, where in 4.1.4 we represented them at the value level with the function `even`.

5.4. Trees

A tree structure is a way to represent hierarchical data. We will work with binary trees in this section, which are trees that contain exactly two sub-trees (nodes). We can define this tree structure using the following implementation:

```
data Tree = Leaf | Node Nat Tree Tree
```

This definition states that a tree is defined as one of:

1. `Leaf`, which has no values
2. `Node`, which holds a number and points to two other trees (which can be either `Node`s or `Leaf`s)

For example, we can use the expression `Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)` to represent the following tree:

```
  2
 / \
1   3
```

Edges can be thought of as the number of "links" from a node to its children. Node 2 in the tree above has two edges: $(2, 1)$ and $(2, 3)$.

Exercise 23

Come up with a few trees by using the value constructors above.

5.4.1. Depth

Definition 9

The depth of a tree is defined as the number of edges from the node to the root.

We can implement the recursive function `depth` as follows:

```
depth : Tree -> Nat
depth Leaf         = 0
depth (Node n l r) = 1 + maximum (depth l) (depth r)
```

If we pass a `Tree` to the function `depth`, then Idris will pattern match the types and we can extract the values. For example, in the `Node` case, we pattern match to extract the sub-trees `l` and `r` for further processing. For the `Leaf` case, since it's just an empty leaf there are no links to it.

In order to prove that the `depth` of a tree is greater than or equal to zero, we can approach the proof as follows:

```
depth_tree_gt_0 : (tr : Tree) -> GTE (depth tr) 0
depth_tree_gt_0 tr = ?prf
```

For the hole, we get:

```
  tr : Tree
--------------------------------------
prf : LTE 0 (depth tr)
```

Doesn't seem like we have enough information. We can proceed with proof by cases:

```
depth_tree_gt_0 : (tr : Tree) -> GTE (depth tr) 0
depth_tree_gt_0 Leaf             = ?prf1
depth_tree_gt_0 (Node v tr1 tr2) = ?prf2
```

Checking the types of the holes:

```
Holes: Main.prf2, Main.prf1
Idris> :t prf1
--------------------------------------
prf1 : LTE 0 0
Idris> :t prf2
  tr1 : Tree
  tr2 : Tree
  _t : Nat
--------------------------------------
prf2 : LTE 0 (S (maximum (depth tr1) (depth tr2)))
```

For the first case, it's pretty easy. We just use the constructor `LTEZero` with implicit `right = 0`:

```
depth_tree_gt_0 : (tr : Tree) -> GTE (depth tr) 0
depth_tree_gt_0 Leaf             = LTEZero {right = 0}
depth_tree_gt_0 (Node v tr1 tr2) = ?prf2
```

For the second case, it also seems we need to use `LTEZero`, but the second argument is `(S (maximum (depth tr1) (depth tr2)))`.

```
depth_tree_gt_0 : (tr : Tree) -> GTE (depth tr) 0
depth_tree_gt_0 Leaf             =
    LTEZero {right = 0}
depth_tree_gt_0 (Node v tr1 tr2) =
    LTEZero {right = 1 + maximum (depth tr1) (depth tr2)}
```

Thus, we have proven that the depth of any tree is greater or equal to zero.

5.4.2. Map and size

We saw how we can use `map` with lists. It would be neat if we had a way to map trees as well. The following definition will allow us to do exactly that:

```
map_tree : (Nat -> Nat) -> Tree -> Tree
map_tree _ Leaf             = Leaf
map_tree f (Node v tr1 tr2) = (Node (f v)
                                   (map_tree f tr1)
                                   (map_tree f tr2))
```

The function `map_tree` accepts a function and a `Tree` and then returns a modified `Tree` where the function is applied to all values of the nodes. In the case of `Leaf`, it just returns `Leaf`, because there's nothing to map to. In the case of a `Node`, we return a new `Node` whose value is applied to the function `f` and then recursively map over the left and right branches of the node. We can use it as follows:

```
Idris> Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)
Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf) : Tree
Idris> map_tree (\x => x + 1) (Node 2 (Node 1 Leaf Leaf)
    (Node 3 Leaf Leaf))
Node 3 (Node 2 Leaf Leaf) (Node 4 Leaf Leaf) : Tree
```

Definition 10

The size of a tree is defined as the sum of the levels of all nodes.

We will now implement `size_tree` which is supposed to return the total count of all nodes contained in a tree:

```
size_tree : Tree -> Nat
size_tree Leaf = 0
size_tree (Node n l r) = 1 + (size_tree l) + (size_tree r)
```

Trying it with a few trees:

```
Idris> size_tree Leaf
0 : Nat
Idris> size_tree (Node 1 Leaf Leaf)
1 : Nat
Idris> size_tree (Node 1 (Node 2 Leaf Leaf) Leaf)
2 : Nat
```

5.4.3. Length of mapped trees

We want to prove that for a given tree and *any* function `f`, the size of that tree will be the same as the size of that tree mapped with the function `f`:

```
proof_1 : (tr : Tree) -> (f : Nat -> Nat) ->
    size_tree tr = size_tree (map_tree f tr)
```

This type definition describes exactly that. We will use proof by cases and pattern match on `tr`:

```
proof_1 Leaf _             = ?base
proof_1 (Node v tr1 tr2) f = ?i_h
```

Checking the types of the holes:

```
Holes: Main.i_h, Main.base
Idris> :t base
  f : Nat -> Nat
--------------------------------------
base : 0 = 0
Idris> :t i_h
  v : Nat
  tr1 : Tree
  tr2 : Tree
  f : Nat -> Nat
--------------------------------------
i_h: S (plus (size_tree tr1) (size_tree tr2)) =
     S (plus (size_tree (map_tree f tr1)) (size_tree (map_tree f tr2)))
```

For the base case, we can just use `Refl`. However, for the inductive hypothesis, we need to do something different. We can try applying the proof recursively to `tr1` and `tr2` respectively:

```
proof_1 : (tr : Tree) -> (f : Nat -> Nat) ->
    size_tree tr = size_tree (map_tree f tr)
proof_1 Leaf _             = Refl
proof_1 (Node v tr1 tr2) f = let IH_1 = proof_1 tr1 f in
                             let IH_2 = proof_1 tr2 f in
                             ?conclusion
```

We get to the following proof state at this point:

```
Holes: Main.conclusion
Idris> :t conclusion
  v : Nat
  tr1 : Tree
  tr2 : Tree
  f : Nat -> Nat
  IH_1 : size_tree tr1 = size_tree (map_tree f tr1)
  IH_2 : size_tree tr2 = size_tree (map_tree f tr2)
--------------------------------------
conclusion : S (plus (size_tree tr1) (size_tree tr2)) =
             S (plus (size_tree (map_tree f tr1))
                 (size_tree (map_tree f tr2)))
```

From here, we can just rewrite the hypothesis:

```
proof_1 (Node v tr1 tr2) f = let IH_1 = proof_1 tr1 f in
                             let IH_2 = proof_1 tr2 f in
                             rewrite IH_1 in
                             rewrite IH_2 in
                             ?conclusion
```

At this point, if we check the type of `conclusion` we will note that we can just use `Refl` to finish the proof.

Conclusion

We've seen how powerful types are. They allow us to put additional constraints on values. This helps with reasoning about our programs since a whole class of non-valid programs will not be accepted by the type checker. For example, if we look at the following function, we immediately know by its type that it returns a natural number:

```
f : Nat -> Nat
f _ = 6
```

Note that there are many constructors (proofs) for `Nat`. For example, we also have that `f _ = 7`. Depending on whether we need 6 or 7 in practice has to be additionally checked. But we're certain that it's a natural number (by the type).

Most programming languages that have a type system have the same expressive power as that of propositional logic (no quantifiers). Dependent types are powerful because they allow us to express quantifiers, which increases the power of expressiveness. As a result, we can write any kind of mathematical proofs.

In mathematics, everything has a precise definition. Same is the case with Idris. We had to give clear definitions of our functions and types prior to proving any theorems.

If Idris proves software correctness, what proves the correctness of Idris?

Trusted Computing Base (TCB) can be thought of as the "axioms" of Idris, that is, we choose to trust Idris and the way it rewrites, inserts a new given, etc.

The most challenging part is to come up with the precise properties (specifications) that we should prove in order to claim correctness for our software.

The beauty of all of this is that almost everything is just about types and finding inhabitants (values) of types.

Further reading

Morris, D. W., Morris, J., *Proofs and Concepts*, 2016

Velleman, J. D., *How to Prove It: A Structured Approach*, 1994

Megill, N., *Metamath*, 1997

Halmos, P., *Naive Set Theory*, 1960

Lipovaca, M., *Learn You a Haskell for Great Good*, 2011

The Idris Community, *Programming in Idris: A Tutorial*, 2015

Wadler, P., *Propositions as Types*, 2014

Pierce, B., *Logical Foundations*, 2011

Pierce, B., *Types and Programming Languages*, 2002

Löh, A., McBride C., Swierstra W., *A tutorial implementation of a dependently typed lambda calculus*, 2010

Martin-Löf, P., *Intuitionistic Type Theory*, 1984

Hofstadter, D., *Gödel, Escher, Bach*, 1979

Sitnikovski, B., *Tutorial implementation of Hoare logic in Haskell*, 2021

Smullyan, R., *The Gödelian Puzzle Book*, 2013

Appendices

Appendix A: Writing a simple type checker in Haskell

This appendix provides a short introduction to the design of type checkers. It is based on the examples of (and may serve as a good introduction to) the book Types and Programming Languages.

Evaluator

Syntax: The syntax per Backus-Naur form is defined as:

```
<term>  ::= <bool> | <num> | If <bool> Then <expr> Else <expr> | <arith>
<bool>  ::= T | F | IsZero <num>
<num>   ::= O
<arith> ::= Succ <num> | Pred <num>
```

For simplicity we represent all of them in a single `Term`:

```haskell
data Term =
    T
    | F
    | O
    | IfThenElse Term Term Term
    | Succ Term
    | Pred Term
    | IsZero Term
    deriving (Show, Eq)
```

Rules of inference

Name	Rule
E-IfTrue	$\frac{v_2 \rightarrow v_2'}{\text{If T Then } v_2 \text{ Else } v_3 \rightarrow v_2'}$
E-IfFalse	$\frac{v_3 \rightarrow v_3'}{\text{If F Then } v_2 \text{ Else } v_3 \rightarrow v_3'}$
E-If	$\frac{v_1 \rightarrow v_1'}{\text{If } v_1 \text{ Then } v_2 \text{ Else } v_3 \rightarrow \text{If } v_1' \text{ Then } v_2 \text{ Else } v_3}$
E-Succ	$\frac{v_1 \rightarrow v_1'}{\text{Succ } v_1 \rightarrow \text{ Succ } v_1'}$
E-PredZero	$\frac{}{\text{Pred O} \rightarrow \text{O}}$
E-PredSucc	$\frac{}{\text{Pred(Succ } v) \rightarrow v}$
E-Pred	$\frac{v \rightarrow v'}{\text{Pred } v \rightarrow \text{ Pred } v'}$
E-IszeroZero	$\frac{}{\text{IsZero O} \rightarrow \text{T}}$
E-IszeroSucc	$\frac{}{\text{IsZero(Succ } v) \rightarrow \text{F}}$
E-IsZero	$\frac{v \rightarrow v'}{\text{IsZero } v \rightarrow \text{ IsZero } v'}$
E-Zero	$\frac{}{O}$
E-True	$\frac{}{T}$
E-False	$\frac{}{F}$

Recall that $\frac{x}{y}$ can be thought of as the implication $x \rightarrow y$ at the metalanguage level, where the actual arrow $\rightarrow$ is the implication at the object level (in this case, simply `eval`).

Given these rules, we will reduce terms by pattern matching on them. Implementation in Haskell is mostly “copy-paste” according to the rules:

```haskell
eval :: Term -> Term
eval (IfThenElse T v2 _) = v2
eval (IfThenElse F _ v3) = v3
eval (IfThenElse v1 v2 v3) = let v1' = eval v1 in IfThenElse v1' v2 v3
eval (Succ v1) = let v1' = eval v1 in Succ v1'
eval (Pred O) = O
eval (Pred (Succ v)) = v
eval (Pred v) = let v' = eval v in Pred v'
eval (IsZero O) = T
eval (IsZero (Succ t)) = F
eval (IsZero v) = let v' = eval v in IsZero v'
eval T = T
eval F = F
eval O = O
eval _ = error "No rule applies"
```

As an example, evaluating `eval $ Pred $ Succ $ Pred O` corresponds to the following inference rules:

```
                ----------- E-PredZero
                pred O -> O
          ----------------------- E-Succ
          succ (pred O) -> succ O
------------------------------------- E-Pred
pred (succ (pred O)) -> pred (succ O)
```

Type checker

Syntax: In addition to the previous syntax, we create a new one for types:

```
<type> ::= Bool | Nat
```

In Haskell:

```haskell
data Type = TBool | TNat
```

Rules of inference: Getting a type of a term expects a term, and either returns an error or the type derived:

```haskell
typeOf :: Term -> Either String Type
```

Name	Rule
T-True	$\dfrac{}{\text{T} : \text{TBool}}$
T-False	$\dfrac{}{\text{F} : \text{TBool}}$
T-Zero	$\dfrac{}{\text{O} : \text{TNat}}$
T-If	$\dfrac{t_1 : \text{Bool}, t_2 : T, t_3 : T}{\text{If } t_1 \text{ Then } t_2 \text{ Else } t_3 : T}$
T-Succ	$\dfrac{t : \text{TNat}}{\text{Succ } t : \text{TNat}}$
T-Pred	$\dfrac{t : \text{TNat}}{\text{Pred } t : \text{TNat}}$
T-IsZero	$\dfrac{t : \text{TNat}}{\text{IsZero } t : \text{TBool}}$

Code in Haskell:

```haskell
typeOf T = Right TBool
typeOf F = Right TBool
typeOf O = Right TNat
typeOf (IfThenElse t1 t2 t3) =
    case typeOf t1 of
        Right TBool ->
            let t2' = typeOf t2
                t3' = typeOf t3 in
                if t2' == t3'
                then t2'
                else Left "Types mismatch"
        _ -> Left "Unsupported type for IfThenElse"
typeOf (Succ k) =
    case typeOf k of
        Right TNat -> Right TNat
        _ -> Left "Unsupported type for Succ"
typeOf (Pred k) =
    case typeOf k of
        Right TNat -> Right TNat
        _ -> Left "Unsupported type for Pred"
typeOf (IsZero k) =
    case typeOf k of
        Right TNat -> Right TBool
        _ -> Left "Unsupported type for IsZero"
```

Going back to the previous example, we can now "safely" evaluate (by type checking first), depending on the type check results.

Environments

Our simple language supports evaluation and type checking but does not allow for defining constants. To do that, we will need some kind of an environment that will hold information about constants.

```haskell
type TyEnv = [(String, Type)] -- Type env
type TeEnv = [(String, Term)] -- Term env
```

We also extend our data type to contain `TVar` for defining variables, and meanwhile also introduce the `Let ... in ...` syntax:

```haskell
data Term =
    ...
    | TVar String
    | Let String Term Term
```

Here are the rules for variables:

Name	Rule
Add binding	$\frac{\Gamma, a : T}{\Gamma \vdash a : T}$
Retrieve binding	$\frac{a : T \in \Gamma}{\Gamma \vdash a : T}$

Haskell definitions:

```haskell
addType :: String -> Type -> TyEnv -> TyEnv
addType varname b env = (varname, b) : env

getTypeFromEnv :: TyEnv -> String -> Maybe Type
getTypeFromEnv [] _ = Nothing
getTypeFromEnv ((varname', b) : env) varname =
    if varname' == varname then Just b else getTypeFromEnv env varname
```

We have the same exact functions for terms:

```haskell
addTerm :: String -> Term -> TeEnv -> TeEnv
getTermFromEnv :: TeEnv -> String -> Maybe Term
```

Rules of inference (evaluator): `eval'` is the same as `eval`, with the following additions:

1. New parameter (the environment) to support retrieval of values for constants
2. Pattern matching for the new `Let ... in ...` syntax

```
eval' :: TeEnv -> Term -> Term
eval' env (TVar v) = case getTermFromEnv env v of
    Just ty -> ty
    _       -> error "No var found in env"
eval' env (Let v t t') = eval' (addTerm v (eval' env t) env) t'
```

We will modify `IfThenElse` slightly to allow for evaluating variables:

```
eval' env (IfThenElse T t2 t3) = eval' env t2
eval' env (IfThenElse F t2 t3) = eval' env t3
eval' env (IfThenElse t1 t2 t3) =
    let t' = eval' env t1 in IfThenElse t' t2 t3
```

The remaining definitions can be copy-pasted.

Rules of inference (type checker): `typeOf'` is the same as `typeOf`, with the only addition to support `env` (for retrieval of types for constants in an env) and the new let syntax.

```
typeOf' :: TyEnv -> Term -> Either String Type
typeOf' env (TVar v) = case getTypeFromEnv env v of
    Just ty -> Right ty
    _       -> Left "No type found in env"
typeOf' env (Let v t t') = case typeOf' env t of
    Right ty -> typeOf' (addType v ty env) t'
    _        -> Left "Unsupported type for Let"
```

For the remaining cases, the pattern matching clauses need to be updated to pass `env` where applicable.

To conclude, the evaluator and the type checker almost live in two separate worlds – they do two separate tasks. If we want to ensure the evaluator will produce the correct results, the first thing is to assure that the type checker returns no error. Another interesting observation is how pattern matching the data type is similar to the hypothesis part of the inference rules. The relationship is due to the Curry-Howard isomorphism. When we have a formula $a \vdash b$ (a implies b), and pattern match on a, it's as if we assumed a and need to show b.

Appendix B: Theorem provers

Metamath

Metamath is a programming language that can express theorems accompanied by a proof checker. The interesting thing about this language is its simplicity. We start by defining a formal system (variables, symbols, axioms and rules of inference) and proceed with building new theorems based on the formal system.

As we've seen, proofs in mathematics (and Idris to some degree) are usually done at a very high level. Even though the foundations are formal systems, it is very difficult to do proofs at a low level. However, we will show that there are such programming languages like Metamath that work at the lowest level, that is formal systems.

The most basic concept in Metamath is the substitution method[30]. Metamath uses an RPN stack[31] to build hypotheses and then rewrites using the rules of inference in order to reach a conclusion. Metamath has a very simple syntax. A token is a Metamath token if it starts with `$`, otherwise, it is a user-generated token. Here is a list of Metamath tokens:

1. `$c` defines constants
2. `$v` defines variables
3. `$f` defines the type of variables (floating hypothesis)
4. `$e` defines required arguments (essential hypotheses)
5. `$a` defines axioms
6. `$p` defines proofs
7. `$=` and `$.` start and end body of a proof
8. `$(` and `$)` start and end code comments
9. `${` and `$}` start and end proof blocks

Besides these tokens, there are several rules:

1. A hypothesis is either defined by using the token `$e` or `$f`

[30] Using this method makes it easy to follow any proof *mechanically*. However, this is very different from understanding the *meaning* of a proof, which in some cases may take a long time of studying.

[31] Reverse Polish Notation is a mathematical notation where functions follow their arguments. For example, to represent $1 + 2$, we would write $1\ 2\ +$.

2. For every variable in `$e`, `$a` or `$p`, there has to be a `$f` token defined, that is any variable in essential hypothesis/axiom/proof must have defined a type
3. An expression that contains `$f`, `$e` or `$d` is active in the given block from the start of the definition until the end of the block. An expression that contains `$a` or `$p` is active from the start of the definition until the end of the file
4. Proof blocks have an effect on the access of definitions, i.e. scoping. For a given code in a block, only `$a` and `$p` remain visible outside of the block

In the following example, we'll define a formal system and demonstrate the use of the rule modus ponens in order to get to a new theorem, based on our initial axioms.

```
$( Declaration of constants $)
$c -> ( ) wff |- I J $.

$( Declaration of variables $)
$v p q $.

$( Properties of variables, i.e. they are well-formed formulas $)
wp $f wff p $. $( wp is a "command" we can use in RPN that represents a\
 well-formed p $)
wq $f wff q $.

$( Modus ponens definition $)
${
    mp1 $e |- p $.
    mp2 $e |- ( p -> q ) $.
    mp  $a |- q $.
$}

$( Definition of initial axioms $)
wI  $a wff I $. $( wI is a "command" that we can use in RPN that repres\
ents a well-formed I $)
wJ  $a wff J $.
wim $a wff ( p -> q ) $.
```

We created constants (strings) `->`, `wff`, etc. that we will use in our system. Further, we defined `p` and `q` to be variables. The strings `wp` and `wq` specify that `p` and `q` are `wff` (well-formed formulas) respectively. The definition of modus ponens says that for a given p (`mp1`) and a given $p \to q$ (`mp2`) we can conclude q (`mp`) i.e. $p, p \to q \vdash q$. Note that outside of this block, only `mp` is visible per the rules above. Our initial axioms state that `I`, `J`, and `p -> q` are well-formed formulas. We separate `wff` from `|-`, because otherwise if we just used `|-` then all of the formulas would be true, which does not make sense.

Having defined our formal system, we can proceed with the proof:

```
${ $( Use block scoping to hide the hypothesis outside of the block $)
    $( Hypothesis: Given I and I -> J $)
    proof_I $e |- I $.
    proof_I_imp_J $e |- ( I -> J ) $.
    $( Goal: Proof that we can conclude J $)
    proof_J $p |- J $=
        wI  $( Stack: [ 'wff I' ] $)
        wJ  $( Stack: [ 'wff I', 'wff J' ] $)
        $( We specify wff for I and J before using mp, since the types \
have to match $)
        proof_I       $( Stack: [ 'wff I', 'wff J', '|- I' ] $)
        proof_I_imp_J $( Stack: [ 'wff I', 'wff J', '|- I', '|- ( I -> \
J )' ] $)
        mp  $( Stack: [ '|- J' ] $)
    $.
$}
```

With the code above, we assume `proof_I` and `proof_I_imp_J` in some scope/context. Further, with `proof_J` we want to show that we can conclude `J`. To start the proof, we put `I` and `J` on the stack by using the commands `wI` and `wJ`. Now that our stack contains `[ 'wff I', 'wff J' ]`, we can use `proof_I` to use the first parameter from the stack to conclude `|- I`. Since `proof_I_imp_J` accepts two parameters, it will use the first two parameters from the stack, i.e. `wff I` and `wff J` to conclude `|- I -> J`. Finally, with `mp` we use `|- I` and `|- I -> J` from the stack to conclude that `|- J`.

Simple Theorem Prover

In this section we'll put formal systems into action by building a proof tree generator in Haskell. We should be able to specify axioms and inference rules, and then query the program so that it will produce all valid combinations of inference in an attempt to reach the target result.

We start by defining our data structures:

```haskell
-- | A rule is a way to change a theorem
data Rule a = Rule { name :: String, function :: a -> a }
-- | A theorem is consisted of an axiom and list of rules applied
data Thm a = Thm { axiom :: a, rulesThm :: [Rule a], result :: a }
-- | Proof system is consisted of axioms and rules between them
data ThmProver a = ThmProver { axioms :: [Thm a], rules :: [Rule a] }
-- | An axiom is just a theorem already proven
mkAxiom a = Thm a [] a
```

To apply a rule to a theorem, we create a new theorem whose result is all the rules applied to the target theorem. We will also need a function that will apply all the rules to all consisted theorems:

```haskell
thmApplyRule :: Thm a -> Rule a -> Thm a
thmApplyRule thm rule = Thm (axiom thm) (rulesThm thm ++ [rule])
    ((function rule) (result thm))

thmApplyRules :: ThmProver a -> [Thm a] -> [Thm a]
thmApplyRules prover (thm:thms) = map (thmApplyRule thm) (rules prover)
    ++ (thmApplyRules prover thms)
thmApplyRules _ _ = []
```

In order to find a proof, we search through the theorem results and see if the target is there. If it is, we just return. Otherwise, we recursively go through the theorems and apply rules in order to attempt to find the target theorem.

```haskell
-- | Construct a proof tree by iteratively applying theorem rules
findProofIter :: (Ord a, Eq a) =>
    ThmProver a -> a -> Int -> [Thm a] -> Maybe (Thm a)
findProofIter _ _ 0 _ = Nothing
findProofIter prover target depth foundProofs =
    case (find (\x -> target == result x) foundProofs) of
    Just prf -> Just prf
    Nothing  -> let theorems = thmApplyRules prover foundProofs
        proofsSet = fromList (map result foundProofs)
        theoremsSet = fromList (map result theorems) in
        -- The case where no new theorems are produced, A union B = A
        if (union proofsSet theoremsSet) == proofsSet then Nothing
        -- Otherwise keep producing new proofs
        else findProofIter prover target (depth - 1)
             (mergeProofs foundProofs theorems)
```

Where `mergeProofs` is a function that merges 2 lists of theorems, avoiding duplicates. An example of usage:

```haskell
muRules = [
  Rule "One" (\t -> if (isSuffixOf "I" t) then (t ++ "U") else t)
  , Rule "Two"   (\t ->
    case (matchRegex (mkRegex "M(.*)") t) of
      Just [x] -> t ++ x
      _        -> t)
  , Rule "Three" (\t -> subRegex (mkRegex "III") t "U")
nn, Rule "Four" (\t -> subRegex (mkRegex "UU") t "")
    ]

let testProver = ThmProver (map mkAxiom ["MI"]) muRules in
  findProofIter testProver "MIUIU" 5 (axioms testProver)
```

As a result we get that for a starting theorem `MI`, we apply rule “One” and rule “Two” (in that order) to get to `MIUIU` (our target proof that we’ve specified).

Appendix C: IO, Codegen targets, compilation, and FFI

This section is most relevant to programmers that have experience with programming languages such as C, Haskell, JavaScript. It will demonstrate how Idris can interact with the outside world (IO) and these programming languages.

In the following examples, we will see how we can compile Idris code. A given program in Idris can be compiled to a binary executable or a back-end for some other programming language. If we decide to compile to a binary executable, then the C back-end will be used by default.

IO

IO stands for Input/Output. Examples of a few IO operations are: write to a disk file, talk to a network computer, launch rockets.

Functions can be roughly categorized into two parts: **pure** and **impure**.

1. Pure functions are functions that will produce the same result every time they are called
2. Impure functions are functions that might return a different result on a function call

An example of a pure function is $f(x) = x + 1$. An example of an impure function is $f(x) =$ launch x rockets. Since this function causes side-effects, sometimes the launch of the rockets may not be successful (e.g. the case where we have no more rockets to launch).

Computer programs are not usable if there is no interaction with the user. One problem that arises with languages such as Idris (where expressions are mathematical and have no side effects) is that IO contains side effects. For this reason, such interactions will be encapsulated in a data structure that looks something like:

```
data IO a -- IO operation that returns a value of type a
```

The concrete definition for `IO` is built within Idris itself, that is why we will leave it at the data abstraction as defined above. Essentially `IO` describes all operations that need to be executed. The resulting operations are executed externally by the Idris Run-Time System (or `IRTS`). The most basic IO program is:

```
main : IO ()
main = putStrLn "Hello world"
```

The type of `putStrLn` says that this function receives a `String` and returns an `IO` operation.

```
Idris> :t putStrLn
putStrLn : String -> IO ()
```

We can read from the input similarly:

```
getLine : IO String
```

In order to combine several IO functions, we can use the `do` notation as follows:

```
main : IO ()
main = do
    putStr "What's your name? "
    name <- getLine
    putStr "Nice to meet you, "
    putStrLn name
```

In the REPL, we can say `:x main` to execute the IO function. Alternatively, if we save the code to `test.idr`, we can use the command `idris test.idr -o test` in order to output an executable file that we can use on our system. Interacting with it:

```
boro@bor0:~$ idris test.idr -o test
boro@bor0:~$ ./test
What's your name? Boro
Nice to meet you, Boro
boro@bor0:~$
```

Let's slightly rewrite our code by abstracting out the concatenation function:

```
concat_string : String -> String -> String
concat_string a b = a ++ b

main : IO ()
main = do
    putStr "What's your name? "
    name <- getLine
    let concatenated = concat_string "Nice to meet you, " name
    putStrLn concatenated
```

Note how we use the syntax `let x = y` with pure functions. In contrast, we use the syntax `x <- y` with impure functions.

The `++` operator is a built-in one used to concatenate lists. A `String` can be viewed as a list of `Char`. In fact, Idris has functions called `pack` and `unpack` that allow for conversion between these two data types:

```
Idris> unpack "Hello"
['H', 'e', 'l', 'l', 'o'] : List Char
Idris> pack ['H', 'e', 'l', 'l', 'o']
"Hello" : String
```

Codegen

The keywords `module` and `import` allow us to specify a name of the currently executing code context and load other modules by referring to their names respectively. We can implement our own back-end for a given programming language, for which we need to create a so-called Codegen (`CG`) program. An empty `CG` program would look like this:

```
module IRTS.CodegenEmpty(codegenEmpty) where

import IRTS.CodegenCommon

codegenEmpty :: CodeGenerator
codegenEmpty ci = putStrLn "Not implemented"
```

Since Idris is written in Haskell, the package `IRTS` (Idris Run-Time System) is a Haskell collection of modules. It contains data structures where we need to implement Idris commands and give definitions for how they map to the target language. For example, a `putStr` could map to `printf` in C.

Compilation

We will show how Idris can generate a binary executable and JavaScript code, as well as the difference between total and partial functions and how Idris handles both cases. We define a dependent type `Vect`, which will allow us to work with lists and additionally have the length of the list at the type level:

```
data Vect : Nat -> Type -> Type where
    VNil  : Vect Z a
    VCons : a -> Vect k a -> Vect (S k) a
```

In the code above we define `Vect` as a data structure with two constructors: empty (`VNil`) or element construction (`VCons`). An empty vector is of type `Vect 0 a`, which can be `Vect 0 Int`, `Vect 0 Char`, etc. One example of a vector is `VCons 1 VNil : Vect 1 Integer`, where a list with a single element is represented and we note how the type contains the information about the length of the list. Another example is: `VCons 1.0 (VCons 2.0 (VCons 3.0 VNil)) : Vect 3 Double`.

We will see how total and partial functions can both pass the compile-time checks, but the latter can cause a run-time error.

```
--total
list_to_vect : List Char -> Vect 2 Char
list_to_vect (x :: y :: []) = VCons x (VCons y VNil)
--list_to_vect _ = VCons 'a' (VCons 'b' VNil)

vect_to_list : Vect 2 Char -> List Char
vect_to_list (VCons a (VCons b VNil)) = a :: b :: []

wrapunwrap : String -> String
wrapunwrap name = pack (vect_to_list (list_to_vect (unpack name)))

greet : IO ()
greet = do
    putStr "What is your name? "
    name <- getLine
    putStrLn ("Hello " ++ (wrapunwrap name))

main : IO ()
main = do
    putStrLn "Following greet, enter any number of chars"
    greet
```

We've defined functions `list_to_vect` and `vect_to_list` that convert between dependently typed vectors and lists. Further, we have another function that calls these two functions together. Note how we commented out the total keyword and the second pattern match for the purposes of this example. Now, if we check values for this partial function:

```
Idris> list_to_vect []
list_to_vect [] : Vect 2 Char
Idris> list_to_vect ['a']
list_to_vect ['a'] : Vect 2 Char
Idris> list_to_vect ['a','b']
list_to_vect 'a' (VCons 'b' VNil) : Vect 2 Char
Idris> list_to_vect ['a','b','c']
list_to_vect ['a', 'b', 'c'] : Vect 2 Char
```

It is obvious that we only get a value for the test `['a', 'b']` case. We can note that for the remaining cases the value is not calculated (computed). If we go further and investigate what happens at run-time:

```
Idris> :exec
Following greet, enter any number of chars
What is your name? Hello
Idris> :exec
Following greet, enter any number of chars
What is your name? Hi
Hello Hi
Idris>
```

Idris stopped the process execution. Going one step further, after we compile:

```
boro@bor0:~$ idris --codegen node test.idr -o test.js
boro@bor0:~$ node test.js
Following greet, enter any number of chars
What is your name? Hello
/Users/boro/test.js:177
    $cg$7 = new $HC_2_1$Prelude__List___58__58_($cg$2.$1, new $HC_2_1$P\
relude__List___58__58_($cg$9.$1, $HC_0_0$Prelude__List__Nil));
                                                                      \
                             ^

TypeError: Cannot read property '$1' of undefined
```

```
...
boro@bor0:~$ node test.js
Following greet, enter any number of chars
What is your name? Hi
Hello Hi
```

We get a run-time error from JavaScript. If we do the same with the C back-end:

```
boro@bor0:~$ idris --codegen C test.idr -o test
boro@bor0:~$ ./test
Following greet, enter any number of chars
What is your name? Hello
Segmentation fault: 11
boro@bor0:~$ ./test
Following greet, enter any number of chars
What is your name? Hi
Hello Hi
```

It causes a segmentation fault, which is a run-time error. As a conclusion, if we use partial functions then we need to do additional checks in the code to cover the cases for potential run-time errors. Alternatively, if we want to take full advantage of the safety that the type system offers, we should define all functions as total.

By defining the function `list_to_vect` to be total, we specify that every input has to have an output. All the remaining checks are done at compile-time by Idris and with that, we're guaranteed that all callers of this function satisfy the types.

Foreign Function Interface

In this example, we'll introduce the FFI system, which stands for Foreign Function Interface. It allows us to call functions written in other programming languages.

We can define the file `test.c` as follows:

```c
#include "test.h"

int succ(int i) {
    return i+1;
}
```

Together with `test.h`:

```c
int succ(int);
```

Now we can write a program that calls this function as follows:

```idris
module Main

%include C "test.h"
%link C "test.o"

succ : Int -> IO Int
succ x = foreign FFI_C "succ" (Int -> IO Int) x

main : IO ()
main = do x <- succ 1
    putStrLn ("succ 1 =" ++ show x)
```

With the code above we used the built-in function `foreign` together with the built-in constant `FFI_C` which are defined in Idris as follows:

```
Idris> :t foreign
foreign : (f : FFI) -> ffi_fn f -> (ty : Type) -> {auto fty : FTy f [] \
ty} -> ty
Idris> FFI_C
MkFFI C_Types String String : FFI
```

This can be useful if there's a need to use a library that's already written in another programming language. Alternatively, with IRTS we can export Idris functions to C and call them from a C code. We can define `test.idr` as follows:

```
nil : List Int
nil = []

cons : Int -> List Int -> List Int
cons x xs = x :: xs

show' : List Int -> IO String
show' xs = do { putStrLn "Ready to show..." ; pure (show xs) }

testList : FFI_Export FFI_C "testHdr.h" []
testList = Data (List Int) "ListInt" $ Fun nil "nil" $ Fun cons "cons" \
$ Fun show' "showList" $ End
```

Running `idris test.idr --interface -o test.o` will generate two files: `test.o` (the object file) and `testHdr.h` (the header file). Now we can input the following code in some file, e.g. `test_idris.c`:

```
#include "testHdr.h"

int main() {
    VM* vm = idris_vm();
    ListInt x = cons(vm, 10, cons(vm, 20, nil(vm)));
    printf("%s\n", showList(vm, x));
    close_vm(vm);
}
```

We will now compile and test everything together:

```
boro@bor0:~$ ${CC:=cc} test_idris.c test.o `${IDRIS:-idris} $@ --includ\
e` `${IDRIS:-idris} $@ --link` -o test
boro@bor0:~$ ./test
Ready to show...
[10, 20]
```

With this approach, we can write verified code in Idris and export its functionality to another programming language.

Appendix D: Implementing a formal system

So far we have been mostly using a set of formal systems to prove software correctness. In this appendix, we show how to *both* create and use a formal system in order to be able to prove facts. For that, we will provide a minimal implementation of Propositional logic, as described in chapter 2.

Here's the syntax of the formal system expressed in BNF, and the code in Idris:

```
--prop   ::= P | Q | R | prop brelop prop
--brelop ::= "&&" | "->"
data Prop = P | Q | R | And Prop Prop | Imp Prop Prop
```

We now have a way to represent some logical formulas, e.g. $P \land Q$ as `And P Q`. Further, in our implementation, we also need a way to differentiate between well-formed formulas and theorems since not all well-formed formulas are theorems. For that, we provide the `Proof` data type and a way to extract a proof:

```
data Proof a = MkProof a
total fromProof : Proof a -> a
fromProof (MkProof a) = a
```

Note that `MkProof (And P Q)` ($\vdash P \land Q$) is different from `And P Q` ($P \land Q$). However, the `MkProof` constructor mustn't be used directly; proofs should only be constructed given the rules that we provide next.

```
-- A, B |- A /\ B
total ruleJoin : Proof Prop -> Proof Prop -> Proof Prop
ruleJoin (MkProof x) (MkProof y) = MkProof (And x y)
-- A, B |- A
total ruleSepL : Proof Prop -> Proof Prop
ruleSepL (MkProof (And x y)) = MkProof x
ruleSepL x = x
```

Another powerful rule is the *Implication Rule*. It accepts a non-proven term `Prop`, whereas other rules accept proven terms; the hypothesis needn't be necessarily true, it only states that “If this hypothesis were a theorem, then that would be a theorem”. The second argument is a function (`Proof Prop -> Proof Prop`) that accepts a `Proof` and returns a `Proof`; basically, another rule which will be used to transform the hypothesis `x`. As a result, it produces the theorem $x \to y(x)$.

```
total ruleImplication : Prop -> (Proof Prop -> Proof Prop) -> Proof Prop
ruleImplication x f = f (MkProof x)
```

For example, we can prove that $\vdash P \land Q \to P \land P$ with `ruleImplication (And P Q) (\x => ruleJoin (ruleSepL x) (ruleSepL x))`.

In Idris we get construction of well-formed formulas for free due to algebraic data structures. In some untyped programming languages, such as Python, we could use hashmaps to simulate the types. In the following code we have to be extra careful with the conditional checks, whereas in Idris it is simpler due to pattern matching.

```
def And(x, y): return {'x': x, 'y': y, 'type': 'and'}
def Imp(x, y): return {'x': x, 'y': y, 'type': 'imp'}
def MkProof(x): return {'v': x, 'type': 'proof'}
def fromProof(x): return x['v'] if isinstance(x, dict) and 'v' in x els\
e None
def ruleJoin(x, y):
  if not isinstance(x, dict) or not isinstance(y, dict) or not 'type' i\
n x or not 'type' in y or x['type'] != 'proof' or y['type'] != 'proof':\
 return None
  return MkProof(And(fromProof(x), fromProof(y)))
def ruleSepL(x):
  if not isinstance(x, dict) or not 'type' in x or x['type'] != 'proof'\
: return None
  wff = fromProof(x)
  if wff['type'] == 'and': return MkProof(wff['x'])
  return MkProof(wff)
def ruleImplication(x, y): return MkProof(Imp(x, fromProof(y(MkProof(x)\
)))) if isinstance(x, dict) and 'type' in x and callable(y) else None
```

Note how we embedded a formal system (Propositional logic) within a formal system (Idris/Python), and we were able to reason about it through symbolic manipulation.

About the author

Boro Sitnikovski has over 10 years of experience working professionally as a Software Engineer. He started programming using the Assembly programming language on an Intel x86 at the age of 10. While in high school, he won several prizes for competitive programming, varying from fourth, third, and first place.

He is an Informatics graduate; his Bachelor's thesis was titled "Programming in Haskell using algebraic data structures", and his Master's thesis was titled "Formal verification of Instruction Sets in Virtual Machines". He has also published papers on software verification. Other research interests include programming languages, mathematics, logic, algorithms, and writing correct software.

He is a strong believer in the open-source philosophy and contributes to various open-source projects.

In his spare time, he enjoys time with his family.