A Tutorial on Type Theory and the LaTTe proof assistant

Frédéric Peschanski (fredokun)

Contents

Introduction

```
(ns latte-tutorial.ch01-front-matter)
```

This document is a tutorial introduction to the LaTTe proof assistant.

My primary goal is to write:

- a quick startup guide to LaTTe,
- a pedagogical overview of its main features,
- a first experiment with important proof rules.

Some non-goals are:

- a complete course about e.g. type theory or logic, or whatever,
- an exhaustive manual for the proof assistant,
- a maths lecture.

Thus I'll go straight-to-the-point, often omitting (probably) important aspects.

The tutorial is heavily inspired by a talk about LaTTe that I gave at EuroClojure'2016. LaTTe changed quite a bit since then, but the video is still a good example of what I call "live coding mathematics".

The source for this tutorial is a *literate programming* document, which means it is both a textual document with Clojure (and LaTTe) code examples, and also a set of heavy commented Clojure source files that can be compiled, executed, tested, etc.

For each namespace `nsp` (in the Clojure terminology), you'll find :

- one file `nsp.clj` containing the Clojure **code source view** with all the explanations (e.g. this very sentence) as comments
- a corresponding file `nsp.clj.md` containing the **document view** as a markdown text with all the source code.

For example, the following line is an expression of the Clojure language that can be evaluated directly when loading the `.clj` source file:

```
((fn [x] x) 42)
;; => 42
```

For many examples the expected evaluation results is shown as a comment: `;; => <value>`.

Remark: The translation of the code source view to the document view is handled by a very simple *Markownize* tool that you can find at: https://github.com/fredokun/markdownize

Also, there are a few exercises and questions in the `master` branch of the tutorial. The solutions are available in the `solutions` branch (which should be at least one commit beyond `master`).

Author

I think it is good to introduce oneself when explaining things to other people.

My name is Frederic Peschanski, I am an associated professor in computer science at Sorbonne University in Paris, France. I do research mostly on theoretical things, but my main hobby is programming thus I try sometimes to mix work and pleasure by coding research prototypes (in various languages, Clojure among them of course).

On my spare time I develop largely experimental free (as in freedom) software, and for a few of them I do get some users which thus involves some maintenance.

You can reach me professionally at the following address: frederic.peschanski@lip6.fr. I will be happy to answer but don't mind to much if it takes some time.

Acknowledgments

I developed LaTTe after reading (devouring, more so) the following book:

> **Type Theory and Formal Proof: an Introduction**.
>
> *Rob Nederpelt and Herman Geuvers*
>
> Cambridge University Press, 2012

That's a heavily recommended lecture if you are interested in logic in general, and the λ-calculus in particular. It is also the best source of information to understand the underlying theory of LaTTe. However it is not a prerequisite for learning and using LaTTe.

I would also like to thank Yeonathan Sharvit and Hiram Madelaine as well as the (few) contributors to LaTTE. And of course "big five" to the Clojure core dev. team and all the community.

About LaTTe

LaTTe stands for "Laboratory for Type Theory Experiments" but it is in fact a perfectly usable **proof assistant** for doing formal mathematics and logic on a computer. It's far from being the most advanced proof assistance technology but it still provides some interesting features.

Remark: The double *TT* of LaTTe intentioanally looks like Π the Greek capital letter Pi, which is one of the very few constructors of the type theory used by LaTTe. This will be explained, at least superficially, in the tutorial.

The basic activity of a proof assistant user is to:

- state "fundamental truths" as **axioms**,
- write **definitions** of mathematical concepts (e.g. what it is to be a bijection)
- state properties about these concepts based on their definition, in the form of **theorem** (or lemma) statements
- and for each theorem (or lemma) statement, assist in writing a **formal proof** that it is, indeed, a theorem.

So it's of no surprise that these are the main features of LaTTe. But it is also a library for the Clojure programming language, which is unlike many other proof assistants designed as standalone tools (one exception being the members of the HOL family, as well as ACL2, both important sources of inspiration). Thanks to the power of the Lisp language in general, and its Clojure variant in particular, all the main features of the proof assistant are usable directly in Clojure programs or at the REPL (Read-Eval-Print-Loop).

Also this means that any Clojure development environment (e.g. Cider, Cursive) can be used as a proof assistant GUI. And this is where most of the power of LaTTe lies. Indeed, the Clojure IDEs support very advanced interactive features. Moreover, one can extend the assistant directly in Clojure and without any intermediate such as a plugins system or complex API. ;; In fact, you develop mathematics in Clojure *exactly like* you develop programs in general, there's not difference at all!

Moreover, the mathematical contents developed using LaTTe can be distributed using the very powerful Clojure ecosystem based on the Maven packaging tool. Also, there is a simple yet effective *proof certification* scheme that corresponds to a form of compilation for the distributed content. Proving things in type theory can become rather computationally intensive, but a certified proof can be "taken for granted".

Last but not least, the main innovative feature of LaTTe is its *declarative proof language*, which makes the proofs follow the *natural deduction* style. The objective is to make LaTTe proofs quite similar to standard mathematical proofs, at least strucutrally. One still has to deal with the Clojure-enhanced Lisp notation, i.e. a perfectly non-ambiguous mathematical notation, only slightly remote from "mainstream" mathematics.

Intended audience

You might be interested in LaTTe because as a Clojure developer you are curious about the lambda-calculus, dependent types, the Curry-Howard correspondance, or simply formal logic and mathematics.

You might also be interested in LaTTe to develop some formal mathematical contents, based on an approach that is not exactly like other proof assistants. I very much welcome mathematical contributions to the project!

Finally, you might be interested in how one may embed a *domain-specific language*, the Lisp way, in Clojure (thus with an extra-layer of data-orientation). Clojure the language is still there when using LaTTe, but you're doing not only programming but also mathematics and reasoning… The *same* difference, the Lisp way…

Or maybe you're here and that's it!

In any case you are very **Welcome**!

Prerequisites

Since LaTTe is a Clojure library, it is required to know at least a bit of Clojure and one of its development environment to follow this tutorial. LaTTe works pretty well in Cider or Cursive, or simply with a basic editor and the REPL.

Note that beyond basic functional programming principles, there's nothing much to learn on the Clojure side. Still, if you don't know Clojure I can only recommend to read the first few chapters of an introductory Clojure book.

License

Tutorial plan

The following steps should be followed in order:

1. first steps (install & friends)
2. the rules of the game (a.k.a. lambda, forall and friends)
3. a bit of logic: natural deduction
4. a glimpse of (typed) set theory
5. a sip of integer arithmetics
6. proving in the large: from proof certification to Clojars deploy

```
(println "Let's begin!")
;; => nil
```

First steps

```
(ns latte-tutorial.ch02-first-steps)
```

In this short chapter I will quickly explain how to install LaTTe and start to work with it.

If you are a seasoned Clojure programmer, there is in fact nothing that distinguishes the installation of LaTTe and the installation of any leiningen-powered Clojure library.

Other build tools, such as boot or deps, could be used but I am personally enjoying working with leiningen and find it also a fantastic build tool for mathematical developments !

Step 1: the project structure

I will take the current `latte-tutorial` project as an example. I have created the basic directory structure with leiningen, using something like the following:

```
$ lein new app latte-tutorial
```

(note that the `$` symbol represents the Shell prompt)

I use the `app` template because as you'll see in the final chapter we will write a `main` function in the final chapter of the tutorial.

After some cleanup I get a `latte-tutorial/` root directory with the following contents:

- `project.clj` the leiningen project file
- `src/latte_tutorial/` the source files (`.clj` or `.clj.md` extensions)
- `LICENSE`, `README.md` some auxiliary files

Step 2: the project files

As for any leiningen the `project.clj` file contains the build informations of our project, in particular its dependencies.

At the time of writing the document, the file contains the following informations:

```
(defproject latte-tutorial "0.1.0-SNAPSHOT"
  :description "A gentle introduction to the LaTTe proof assistant."
```

```
  :url "https://github.com/latte-central/latte-tutorial"
  :license {:name "MIT License"
            :url "http://opensource.org/licenses/MIT"}
  :dependencies [[org.clojure/clojure "1.10.0"]
                 [latte "1.0b2"]
                 [latte-prelude "1.0b2"]
                 [latte-sets "1.0b2"]
                 [latte-integers "1.0b2"]])
```

The `:dependencies` key is most interesting. We first require the Clojure implementation itself, of course. And our LaTTe dependencies are as follows:

- The core `latte` implementation, which provides the main user interface of the proof assistant. It itself depends on the `latte-kernel` project which implements the type theory used in LaTTe
- the `latte-prelude` standard library, which provides the fundamental reasoning tools such as propositional operators, equality, quantifiers, etc.
- the `latte-sets` library of (typed) set theory, which we will use in the tutorial
- the `latte-integers` arithmetic library, which we will also discuss

To be sure that everything's fine regarding the dependencies, you might try the following on the command line:

```
$ lein deps
```

In case there is a problem with the dependencies, some error message will be printed out. Otherwise, if nothing happens, you're good to go!

Step 3: coding mathematics

In this final step, we begin our work with LaTTe. As for any Clojure program we simply have to create a namespace source file with the `.clj` extension and start coding.

We will proceed to the file `src/latte_tutorial/ch03_game_rules.clj` corresponding to the namespace `latte-tutorial.ch03-game-rules` in which we'll start coding mathematics.

Let's proceed …

The rules of the game

We are now ready to begin our first mathematical development in LaTTe. This will be in a `ch03-game-rules` namespace whose declaration is as follows:

```
(ns latte-tutorial.ch03-game-rules
  ;; In this namespace we will only play with (lambda-)terms and types,
  ;; so we require only very few top-level forms from the `core` namespace.
  (:require [latte.core :refer [term type-of type-check?]]))
;; => nil
```

Remark: the `nil` returned by the `ns` form is in general quite a good thing. It means that the requirements are satisfied.

Lambda the ultimate

One assumption I make is that you, the reader, you know the "pure" (i.e. untyped) lambda-calculus. This is not a very strong assumption because you can find it right at the core of you favorite programming language. Indeed, in Clojure the `lambda` is called `fn`, which we use to define anonymous functions. Note that `fn` corresponds to (the classical) `lambda` only if it accepts a single parameter.

As a trivial example consider the identity function:

```
(fn [x] x)
```

This function can be applied to anything to yield the very *same* anything:

```
((fn [x] x) 42)
;; => 42

((fn [z] z) :anything)
;; => :anything
```

We also see that the variables, e.g. `x`, `z`, can be renamed without changing the function, it's still the identity function (unless there's some clash in the renaming). In the set of useful functions, this is maybe the simplest of all and it even deserve a name in the standard Clojure library:

```
(identity 42)
;; => 42
```

As a second, slightly more complex example, consider the composition of two functions `f` and `g`:

```
(fn [f] (fn [g] (fn [x] (g (f x)))))
```

Now we can apply this function a few times to see how it works:

```
((((fn [f] (fn [g] (fn [x] (g (f x)))))
   even?)                          ;; (==> int boolean)
  (fn [y] (if y "even" "odd"))) ;; (==> boolean string)
 42)
;; => "even"
```

We took our composition function, and provided the predicate `even?` as the `f` function. As we comment above, this takes an `int` and returns a `boolean`. Then, the `g` function that we use takes a `boolean` and returns a `string`. Thus the composition `g°f` takes an `int` and returns a `string`, either `"even"` if the input of `f` is even, and `"odd"` otherwise.

```
((((fn [f] (fn [g] (fn [x] (g (f x)))))
   even?)                          ;; (==> int boolean)
  (fn [y] (if y "even" "odd"))) ;; (==> boolean string)
 41)
;; => "odd"
```

This is also an important basic functional building block, and it is named `comp` in the Clojure standard library.

```
((comp (fn [y] (if y "even" "odd")) even?) 42)
;; => "even"

((comp (fn [y] (if y "even" "odd")) even?) 41)
;; => "odd"
```

These are roughly the only things you need to know about Clojure to begin exploring the LaTTe features.

Types, really ?

In the composition example above we did something rather unusual in Clojure: we described the type of functions using the notation `(==> T U)` with `T` the type of the input, and `U` the type of the returned value.

As we'll quickly see, in type theory the explicit mentioning of types becomes an essential part of the mathematical language. As I often say if I am not totally sold to the use of (static) types in programming (otherwise I would maybe not use Clojure for starters), I am totally sold to the use of types in logic and mathematics. I find much more interests in having a "set of integers" rather than a "set" without further mentioning of the type of things I'll find as elements.

So if we want to define e.g. the identity function not directly in Clojure but this time in LaTTe (which is also Clojure by the way), we will have to add some type annotations.

In LaTTe, `lambda` is called `lambda`, or `λ`, rather than `fn`. In this part of the tutorial we will rather use the beautiful `λ` rather than its ascii spelling. This is to emphasis

that we are not yet at the user-level of the proof assistant, we only play with its λ-calculus kernel.

Thus our starting point is this:

```
;; The identity function in LaTTe (initial version)
(term
  (λ [x] x))
 --> Unhandled clojure.lang.ExceptionInfo
     Parse error
     {:msg "Wrong bindings in λ form",
      :term (λ [x] x),
      :from {:msg "Binding must have at least 2 elements", :term [x]}}
```

The `term` (or `latte.core/term`) form takes a LaTTe expression in input, parses it, typecheck's it and returns the internal representation of the term as Clojure data. As the exception raised by LaTTe makes clear, there is something missing in our identity function. In fact we need to give an explicit type to the variable `x`, let's try with an arbitrary type named `A`:

```
;; The identity function in LaTTe (second version)
(term
 (λ [x A] x))
 --> Unhandled clojure.lang.ExceptionInfo
     Type checking error
     {:msg "Cannot calculate codomain type of abstraction.",
      :term (λ [x A] x),
      :from {:msg "Cannot calculate type of variable.",
      :term x,
      :from {:msg "No such variable in type context", :term A}}}
```

As you can see LaTTe is quite verbose when something goes wrong, which is rather a good thing when debugging mathematics! Here we have a type error, and the ultimate reason is that the variable `A` is defined nowhere.

What we would like is to make `A` an arbitrary type, so for this we will add an extra layer of abstraction to our λ.

```
;; ;; The identity function in LaTTe (third, correct version)
(term
 (λ [A :type] (λ [x A] x)))
;; => (λ [A *] (λ [x A] x))
```

This time the form evaluates! The `:type` keyword denotes the "type of all types". Hence in LaTTe the (type-)generic identity function first takes an arbitrary type `A` as a first argument, and returns a function that takes an arbibrary `x` of type `A` to finally return `x` itself. That's generic as it can be!

The value returned by the `term` form (technically a Clojure macro) is the internal representation of the terms in LaTTe. It's almost like what is written except that the type of types is written `*` internally. In fact I do not find any (high-quality) unicode font supporting this so I replaced the actual symbol `Eight Spoked Asterisk Emoji` (`U+2733`) by the "standard" asterisk `*`. I've made this choice so that we know when terms are internal vs. written by the user. Note also that if we use the ascii

`lambda` we get the same result:

```
(term
 (lambda [A :type] (lambda [x A] x)))
;; => (λ [A *] (λ [x A] x))
```

The type of lambda

In LaTTe most terms have types, and the identity function of the previous section is no exception. Thus the question is: *what is the type of the following?*

```
(λ [A :type] (λ [x A] x))
```

The answer to this question is perhaps the most important aspect of type theory, and in fact it is a very simple answer:

> The type of λ (`lambda`) is ∀ (`forall`)

You might read somewhere that it is a (kind of) product, but it's much simpler to relate *functions* (constructed with λ) and *universal quantifications* (introduced with ∀).

In general, the typing formula is something like:

```
(type-of (λ [x T] e)) ≡ (∀ [x T] (type-of e))
```

(note that this formula is not actual Clojure code)

Let's try to find the type of the identity function. We have something like:

```
(type-of (λ [A :type] (λ [x A] x)))
≡ (∀ [A :type] (∀ [x A] (type-of x)))
```

Since `x` is supposed of type `A`, we should have:

```
(type-of x) ≡ A
```

Hence:

```
(type-of (λ [A :type] (λ [x A] x)))
≡ (∀ [A :type] (∀ [x A] A))
```

To see if LaTTe agrees with this, we can use the `type-check?` predicate.

```
(type-check?
 ;; is the term:
 (λ [A :type] (λ [x A] x))
 ;; of type:
 (∀ [A :type] (∀ [x A] A))
 ;; ?
 ) ;; and the answer is:
;; => true
```

The proof assistant seems to agree… good!

Note that in case you don't find λ or ∀ on your keyboard, you can alternatively "type" the following (pun intended):

```
(type-check?
 (lambda [A :type] (lambda [x A] x))
 (forall [A :type] (forall [x A] A)))
;; => true
```

Curry Howard Take One: Propositions as Types

We are now ready for the Curry Howard enlightenment, take one. If you have some level of mathematical background, then the `∀` symbol and its spelling as "for all" should sound rather familiar. The logical interpretation is that of *universal quantification*.

Taken literally, the type:

```
(∀ [A :type] (∀ [x A] A))
```

means something like:

> For all `A` of type `:type`, and for all `x` of type `A`, then `A`

That's not very effective as a logical statement. But everything is *here* except it is not *that* apparent. To obtain a simpler reading (of the same type), we first change a little bit the way we talk about the type `A`.

Saying that `A` is of type `:type` is exactly the same as saying that `A` is a type, and as a synonym in type theory:

> `A` is a proposition (expressed as a type)

(*"proposition as type"* anyone?)

Moreover, remark that in the subterm `(∀ [x A] A)` the variable `x` remains *unusued* (technically we say that there is no free occurence of the variable `x` in the body of the quantifier (which is `A` in our example). Informally, this is when the variable remains *"unused"*.

When this is the case, we can use the following simplified notation:

> In a term of the form `(∀ [x A] E)` with an arbitrary type expression `E` if there is no free occurrence of `x` in `E` then the term is the same as: `(==> A E)`

According to this rule, we thus have:

```
(∀ [A :type] (∀ [x A] A))
≡ (∀ [A :type] (==> A A))
```

We can see directly in LaTTe that both notations are considered "the same":

```
(term
 (∀ [A :type] (∀ [x A] A)))
;; => (Π [A *] (==> A A))

(term
```

```
 (∀ [A :type] (==> A A)))
;; => (Π [A *] (==> A A))
```

The internal terms are the same, and in fact the so-called "unparser" of LaTTe automatically applies the simplification.

As a side note, we can see that our universal quantification was rewritten using the capital greek letter Pi (Π). This "new" quantifier is called a **product** and there are some reasons why this terminology is favored over universal quantification in many type theories. But in LaTTe at least there is no difference at all, and the reason we use Π rather than `∀` in the internal representation is like for the asterisk, and also by "historical filiation". Users are encouraged to use the usual mathematical notation for universal quantification.

Hitherto we can propose a better logical meaning for the type of the identity function:

> For all proposition (as-type) `A`, `A` implies `A`

Seen as a logical statement `(==> P Q)` is the implication saying that if `P` holds then also `Q` holds. Hence here, the logical statement is that logical **implication is reflexive**.

Let's try to put the simplified notation into practice:

```
(type-check?
 ;; term:
 (λ [A :type] (λ [x A] x))
 ;; of type:
 (∀ [A :type] (==> A A)))
;; => true
```

It works! The identity function captures the fact that implication is reflexive, nice!

Also, the (non-obviously) equivalent computational interpretation is unsurprising:

> For all type `A`, the identity function for this type takes an argument of type `A` and returns a value of type `A` also

Here we are at the heart of the Curry Howard correspondance. A lambda-abstraction (i.e. a single argument anonyous function) such as identity, has two complementary and in fact equivalent interpretations:

1. a computational interpretation: the arrow type `(==> P Q)` saying that the function takes a value of type `P` as input, and outpus a value of type `Q`.
2. a logical interpretation: the implication `(==> P Q)` saying that if `P` holds then so is `Q`.

This is an important take away!

Exercise

A LaTTe version of the composition function we wrote in Clojure is the following:

```
(term
 ;; in Clojure: (fn [f] (fn [g] (fn [x] (g (f x))
 (λ [A B C :type]
    (λ [f (==> A B)]
```

```
      (λ [g (==> B C)]
         (λ [x A] (g (f x)))))))
;; => (λ [A B C *] ... etc ...
```

Remark: the so-called "telescoped" notation `(λ [A B C :type] ...)` is the same as `(λ [A :type] (λ [B :type] (λ [C :type] ...)` This is a common and useful notational facility (internal lambda's have only one argument).

Questions:

- write the type of the term above (the telescope notation is also avaiable for `∀`). Check your answer with `type-check?` (or *cheat* with what's come next but that's cheating!).
- what is the logical interpretation of this type? Does it say something interesting about implication? (hint: it should!)

Tell me your type!

So we saw terms, e.g. the (type-generic) identity function, and we saw types such as the reflexivity of implication. In most typed programming language there is a clear distinction between:

- the terms that express the dynamics of the programs
- their types that are checked statically at compile-time

In LaTTe as in most type theories there is no such a strong distinction.

- first, types are *also* terms!
- however many terms are *not* types.

So how do we make the distinction now? The first and principal rule is the following:

> Only terms whose type is `:type` (or `*` internally) are actually called *types*

Thus, when we write `(λ [A :type] ...)` we explicitly say that `A` is indeed a type, since it is of type `:type`.

The term `:type` (equivalently `*`) is thus called *"the type of types"*.

But we might wonder if `:type` has itself a type, and in this case what would be this *"type of the type of types*" (sic!). In some old type theories, the type of `:type` was `:type` itself… And in fact it was shown at a much later time that this cyclic definition yields an inconsistent theory: a paradox similar to the problematic notion of a set potentially containing itself.

It is common in modern type theories to introduce an unbounded hierarchy of universe levels. The type of types is called the universe of level 0. The type of the universe 0 is universe 1, etc. However this is in the case of LaTTe too much a complex solution to the problem at hand.

So we are back to the question: what is the type of `:type`, if any?

As it happen, we can ask LaTTe directly, and this is how we ask:

```
(type-of :type)
;; => □
```

The type of a star (*) is thus a square (□)! Well, we know that * is `:type`, the type of all "actual" types. Since the type of `:type` cannot be `:type` (for the sake of logical consistency) it is not an "actual" type and is thus called *a sort*. So we have the rule:

> The type of all types is **the sort** `:type`

The term represented as a square □ is also a sort, and it is called `:kind`. So we have:

> The type of the sort `:type` is the sort `:kind`

Let's check if `:kind` has a type too, now that we know the `type-of` trick:

```
(type-of :kind)
;; --> Unhandled clojure.lang.ExceptionInfo
;;     Type checking error
;;     {:msg "Kind has not type", :term □}
```

How unfriendly, we got an exception! But at least the message is clear:

> The sort `:kind` has no type!

In fact `:kind` (or □) is the *only* term in LaTTe that has no type. This is thanks to this "escape hatch" that we avoid the inconsistency of cyclic definitions, and the complexity of universe levels.

The fact that we do not need more "machinery" in LaTTe is because it is a very simple type theory, much simpler than the ones found in most other proof assistants. And unlike most proof assistants, the simple type theory used in LaTTe offers a decisive and distinguishing feature

> It is decidable, and quite efficient, to compute the type of an arbitrary (well-formed) term

This is what the `type-of` form allows us to do: ask the type of a term. Let's try on the identity function:

```
(type-of (λ [A :type] (λ [x A] x)))
;; => (Π [A *] (==> A A))
```

The returned value should be a type then, so let's check this:

```
(type-of (Π [A *] (==> A A)))
;; => *
```

Yes! it's a type since we obtain the sort `:type`.

The `type-of` form decides what is called the *"type synthesis"* problem, and we will see at a later stage that it's quite a useful component of our logical toolbox.

Exercise : compute the type of the composition function using `type-of` (so *now* you are encouraged to "cheat"!)

Curry Howard Take Two: Programs as Proofs

We are now ready for the Curry Howard enlightenment, take two.

With what we learned in the previous sections, we are now able to build implications and universal quantifications using lambda's, but we are missing one important piece of the puzzle: a way to *use* an implication or universal quantification to perform a *deduction*.

In the so-called *natural deduction*, an inference rule called the *modus ponens* explains how an implication can be used in a reasoning step. In plain english, the rule states:

> If we know that "A implies B", and if moreover "A holds". Then we can deduce that "B holds" too.

This proposition, which is undoubtedly among the most important rules of logic, can be rephased as a type:

```
(==> (==> A B) A
     B)
```

Note that the outermost implication is used here as a kind of "trailed" conjunction: if we *first* have `(==> A B)` and *then* we have `A`, *ultimately* we can deduce `B`.

In the next chapter we will define the more traditional conjunction (`and`) operator, but in type theory the "trailed" implication is used most of the time since it is primitive.

According to natural deduction (and logical reasoning indeed), the *modus ponens* rule should be *true*. This could be a basic law of our logic, thus *modus ponens* would be true *philosophically* (more precisely, *axiomatically*). But if philosophical truth is OK, mathematical proof is best! The question thus becomes: *what is a proof?* If in philosophy and "everyday" mathematics this question can bring us quite far, once again type theory offers a very simple and "natural" answer: a *proof* is simply a *program* expressed as a lambda-term, and whose type is the theorem we want to prove. This is the programs-as-proofs part of the Curry-Howard correspondance.

In LaTTe, this means that in order to prove a proposition P, we have to find a term t whose type is P. Note that we are not trying to find a particular program/term t but *any* term t whose type is P will do. This aspect, named *proof irrelevance*, makes proving with programs slightly different from programming *per se*. However some proofs are more elegant than others, more efficient (i.e. smaller) than others, etc. So this is not totally unlike programming when before reaching for an efficient solution, one often begins with a naive solution.

Going back to our *modus ponens* proposition, we are thus trying to fill the star symbol below with an adequate term:

```
(type-check?
 [A :type] [B :type]
 ;; find a term ...
 *
 ;; of type
 (==> (==> A B) A
```

```
      B))
;; => false  (for the moment)
```

To avoid having to many nested lambda's, we introduce a *context* (akin to a lexical environment in programming terms) composed of two arbitrary variables: `A` and `B` both of type `:type`, hence arbitrary propositions. We could have introduced them using universal quantifiers but we would have lost the core of what we want to prove. Similarly in Clojure we write:

```
(defn myfun [A B] ...)
```

rather than:

```
(def myfun (fn [A] (fn [B] ...)))
```

Let's go back to our `type-check?` exercise. We know that an implication is built using a lambda abstraction, i.e.:

> A type of the form `(==> X Y)` is built using a term of the form `(λ [v X] e)` with `e` a term of type `Y`, which in most cases must use the variable `v` declared of type `X`.

Our implication has three members so this becomes:

> A type of the form `(==> X Y Z)` is built using a term of the form `(λ [v X] (λ [w Y] e))` with `e` a term of type `Z`, using the variable `v` declared of type `X`, and `w` of type `Y`.

The inner implication `(==> A B)` resembles a *function type*: a function from type `A` to type `B`. Let's called this function `f`. And then we have a variable of type `A`: let's call it `x`. Hence we could use something of the form:

```
(λ [f (==> A B)]
  (λ [x A]
      ? ;; should be of type B
))
```

Let's think in terms of programs now. We have a function `f` from `A` to `B` and a variable `x` of type `A`. Applying `f` to `x`, hence `(f x)`, should produce a value of type `B`, exactly what we need. Let's try this!

```
(type-check?
 [A :type] [B :type]
 ;; is the term ...
 (λ [f (==> A B)]
    (λ [x A]
       (f x)))
 ;; of type
 (==> (==> A B) A
      B) ;; ?
 )
;; => true  (it works!)
```

We just wrote a formal proof of the *modus ponens*, and it was like a very basic typed functional program!

An very important take away is the tight relation between:

- *computation* with function application on the one side, and
- *logic* on the other side, with *modus ponens* a.k.a. deduction.

Universal quantification

In type theory, implication is a special case of universal quantification, both are introduced by lambda's. So it is not a surprise that using a universally quantified proposition is *also* related to function application.

To demonstrate this, we will take a typical example from the logic curriculum:

> All mens are mortal. Socrates is a man. Hence Socrates is mortal.

This is a *syllogism* attributed to Aristotle. We are very far from modern logic but at least we can use this to illustrate the instantiation of universal quantification: the particular Socrates is an instantiation of "all men".

We can translate the proposition in LaTTe, e.g. as follows:

```
(type-check?
 [Thing :type] [man (==> Thing :type)] [mortal (==> Thing :type)]
 [socrates Thing]
 ;; we want to replace the following by a term ...
 *
 ;; of type
 (==> (∀ [t Thing]
        (==> (man t) (mortal t))) ;; all men are mortal
      (man socrates)  ;; socrates is a man
      ;; thus
      (mortal socrates))) ;; socrates is mortal
;; => false  (for now ...)
```

Once again we have an implication hence the term we are looking for is of the form:

```
(λ [H1 (∀ [t Thing]
          (==> (man t) (mortal t)))]
  (λ [H2 (man socrates)]
     ?  ;; should be a term of type  (mortal socrates)
))
```

In the first step, we can build a term of type `(==> (man socrates) (mortal socrates))` by instantiating the universal quantification on "all things" using the particular `socrates`, hence the term: `(H1 socrates)`. Our "puzzle" then becomes:

```
(λ [H1 (∀ [t Thing]
          (==> (man t) (mortal t)))]
  (λ [H2 (man socrates)]
     ((H1 socrates)
          ?  ;; should be a term of type  (man socrates)
)))
```

The term `H2` has exactly the required type, hence in the final step we have:

```
(type-check?
 [Thing :type] [man (==> Thing :type)] [mortal (==> Thing :type)]
 [socrates Thing]
 ;; the term ...
 (λ [H1 (∀ [t Thing]
          (==> (man t) (mortal t)))]
    (λ [H2 (man socrates)]
       ((H1 socrates)
         H2)))
 ;; is of type
 (==> (∀ [t Thing]
        (==> (man t) (mortal t)))
      (man socrates)
      (mortal socrates)))
;; => true  (yes!)
```

We thus wrote a formal proof that Aristotle's syllogism is indeed a theorem in type theory and LaTTe.

The rules of the game, in summary

1. the type of a lambda-abstraction is a universal quantification
2. if the universally quantified variable is unused, then we have a logical implication, which is *also* an arrow type from the computational point of view
3. function application is instantiation of universal quantification, it is *also* logical deduction in the case of implication
4. types are propositions (a.k.a. Curry Howard take one)
5. programs are proofs (a.k.a. Curry Howard take two)
6. the type of `:type` is the sort `:kind`, and kind is the only term without a type

These are the basic rules, but we are still far from being able to do actual mathematical developments using only these. Indeed, directly writing lamda-terms to prove mathematical statements quickly becomes cumbersome and too far away from the way proofs are generally expressed in mathematics. Moreover, we need higher level abstractions, so in the next chapter we will meet the actual LaTTe proof assistant.

Bits of logic with natural deduction

Now that we have some knowledge of the rules of the game, we can start playing with the LaTTe proof assistant, and do some actual logic.

```
(ns latte-tutorial.ch04-logic-bits
  ;; In this namespace we will start to use LaTTe "for real",
  ;; so we introduce the main forms from the core namespace
  (:require [latte.core :as latte :refer [definition defthm deflemma
                                          example try-example
                                          proof try-proof
                                          assume have pose qed
                                          forall lambda]]
            ;; we will also exploit the basic proposition from the prelude
            [latte-prelude.prop :as p :refer [and and* or or* not <=>]]

            ;; and existentials require the following namespace
            [latte-prelude.quant :as q :refer [exists ex ex-def]]

            ;; we will also use the documentation
            [clojure.repl :refer [doc]])


  ;; also, these belong to logic or formal arithmetic
  (:refer-clojure :exclude [and or not]))
```

Natural deduction is, very roughly, a way of presenting and formalizing logics based on:

- introduction rules, or how to construct logical statements
- elimination rules, or how to take them apart.

We already encountered introduction and elimination rules that are in a way primitive in type theory and LaTTe:

- the introduction of a universal quantifier - or an implication as a special case - using a lambda abstraction
- the elimination of a universal quantifier or an implication using function application.

In this chapter of the tutorial, we will discuss the introduction and elimination rules for other important logical constructs. The implementation of this basic rules can be found in the prop and quant namespaces of the latte-prelude library, a.k.a. the

"standard" library.

Conjunction

Conjunction, or logical `and`, is where most presentations of natural deduction start, so let's follow the tradition.

The definition is provided in the prelude:

```
(definition and
   "logical conjunction."
   [[A :type] [B :type]]
   (forall [C :type]
           (==> (==> A B C)
                C)))
```

But we will write our own version so that we can "play" with it (the one defined in the prelude is declared *opaque* which means the definition is hidden from the user).

```
(definition my-and
  "logical conjunction, a remake"
  [[A :type] [B :type]]
  (forall [C :type]
    (==> (==> A B C)
         C)))
;; => [:defined :term my-and]
```

A mathematical definition in LaTTe is of the form:

```
(definition <name>
   "<docstring>"
   [[x1 T1] ... [xN TN]] ;; <parameters>
   <body-type>)
```

The `<body>` of the definition is a type, hence a term of type `:type` that is parameterized. Such a definition is also a `def` in the Clojure sense, it has e.g. a documentation:

```
(doc my-and)
```

```
latte-tutorial.ch04-logic-bits/my-and
([[A :type] [B :type]])

(forall [C :type] (==> (==> A B C) C))
```

Definition: logical conjunction, a remake.

We will see many definitions in this tutorials, so let's go back to what is called the "second order characterization of conjunction" (originating from System F). While this can be interpreted as a generic proof scheme based on the knowledge of two propositions `A` and `B`, in natural deduction we prefer to decompose the scheme in two parts: the introduction of a conjunction, or its eliminations.

Introduction rule

Informally, the introduction rule for conjunction is often presented as follows:

```
   A     B
============= (and-intro)
  (and A B)
```

The horizontal "double bar" means implication, thus this reads as follows:

> If both proposition `A` and `B` hold, then we can deduce that the conjunction `(and A B)` also holds.

This is often introduced in an axiomatic way, i.e. without any justification, but in type theory the definition above can be used to prove this as a theorem. To introduce a theorem, we use the `defthm` form, and this is how we can formalize the introduction rule in LaTTe:

```
(defthm my-and-intro
  "Introduction rule for conjunction."
  [[A :type] [B :type]]
  (==> A B
       (my-and A B)))
;; => [:declared :theorem my-and-intro]
```

As LaTTe explains us, the theorem is now *declared*, however we cannot use it because we first have to prove that the theorem indeed *is* a theorem.

Note that the internal representation of the theorem is not hidden from the user, and we can find a few interesting informations by inspecting it.

```
my-and-intro
;; => #latte_kernel.defenv.Theorem{
;;       :name my-and-intro,
;;       :params [[A *] [B *]],
;;       :arity 2,
;;       :type (Π [⇧ A]
;;                (Π [⇧ B]
;;                   (#'latte-tutorial.ch04-logic-bits/my-and A B))),
;;       :proof nil}
```

In the Clojure terminology this is a *record* with a few self-explicit fields. We can see that the `:proof` field is `nil`. To understand that "everything is under control", let's try to use the theorem somehow.

```
(try-example [[A :type]]
    (==> A A (my-and A A))
  (qed (my-and-intro A A)))
;; = > [:failed "Proof failed: Qed step failed: cannot infer term type."
;;              {:cause {:msg "Theorem has no proof.",
;;                       :thm-name my-and-intro},
;;               :meta {:line 165, :column 3}}]
```

The `try-example` form let us state propositions together with their proof. It is like

a theorem that we do not plan to save for further use. In real developments, the use of the `example` variant is recommended since it will throw an exception in case of a problem, hence it is a nice utility for both documenting by example as well as self-testing. We will go back to the way the proofs are written at a later stage, so let's just observe that using a theorem without a proof is forbidden in LaTTe (or at least it should be).

Thus we have to write a formal proof that `my-and-intro` really is a theorem. There are two options in LaTTe to do so:

1. we can write a proof term, using the proof-as-program side of the Curry-Howard correspondance, like we did previously
2. or we can write a declarative proof following the natural deduction style.

We already know how to write a proof term, so let's first demonstrate the first possibility. To prove a theorem in LaTTe, we have to use the `proof` function (it is not a macro because it doesn't have to be), which expects arguments of the following form:

```
(proof '<theorem-name>
   <proof-script>)
```

The `<proof-script>` part is based on a very reduced set of constructions, the simplest one being `(qed <proof-term>)` when we want to conclude the proof using a proof term. Here is an example of a direct demonstration using `qed`:

```
(proof 'my-and-intro
  (qed (λ [x A]
          (λ [y B]
             (λ [C :type]
                (λ [z (==> A B C)]
                   ((z x) y)))))))
;; => [:qed my-and-intro]
```

The proof is accepted but even for this simple fact writing explicitly the whole proof term is cumbersome at the very least and, most of all, quite unlike a "pencil & paper" mathematical proof. So we will rewrite this proof using a more *declarative* style. We will also use the variant `try-proof` which doesn't generate an exception when a proof fails. This is a very useful tool when elaborating proofs step by step.

First we have our basic assumptions: that the propositions `A` and `B` hold. We can write the following:

```
(try-proof 'my-and-intro
  (assume [x A
           y B]))
;; => [:failed my-and-intro {:msg "Proof incomplete."}]
```

Assumptions as introduced by the following form:

```
(assume [hyp1 typ1
         ...
        [hypN typN]
   <script-within-assumptions>)
```

Technically speaking, this creates the enclosing lambda's corresponding to the as-

sumptions. But from a logical point of view, we just have to see this as a statement of assumptions.

When evaluating the form above the "error" message we get is that the proof is incomplete. This is good because it also says that there is no error in our reasoning, only we're not finished.

To finish the proof we have to find a term of type `(my-and A B)`. Now, let's remind the internal definition of conjunction:

```
(forall [C :type] (==> (==> A B C) C))
```

There are two further assumptions here: that we have an arbitrary type `C` and an assumption, let's say `H` that `(==> A B C)` (from `A` and `B` we can deduce `C`). So in the next step we get the following:

```
(try-proof 'my-and-intro
  (assume [x A
           y B]
    (assume [C :type
             H (==> A B C)])))
;; => [:failed my-and-intro {:msg "Proof incomplete."}]
```

Note that we could use a single `assume` form here but we have here perhaps a cleaner separation between the theorem assumptions (often called the *hypotheses*), and the ones corresponding to the definition of `my-and`. In the next step, we have to build, from all these assumptions, a term of type `C`. We say that our current *goal* is the proposition `C`. Let's proceed step-by-step. First, we can see `H` as a function that expects a first argument of type `A`, and returns a function of type `(==> B C)`. We can use the `have` form to create such an intermediate result.

```
(try-proof 'my-and-intro
  (assume [x A
           y B]
    (assume [C :type
             H (==> A B C)]
      (have <a> (==> B C) :by (H x)))))
;; => [:failed my-and-intro {:msg "Proof incomplete."}]
```

A `have` proof step has the following form:

```
(have <step-name> <prop/type> :by <proof-term>)
```

This an intermediate proof that the proposition `<prop/type>` (remember proposition-as-type) holds with the given `<proof-term>`. Hence this is like a statement and proof of an intermediate lemma, a building block for a full proof.

Once again there is no error here, and what we did is to simultaneously:

- from a computational point of view created a function named `<a>` of type `(==> B C)`
- from a logical point of view demonstrated that `(==> B C)` holds, under the specified hypotheses.

This is really the Curry-Howard correspondance at work. Not that such an intermediate result is checked by LaTTe. To observe this, let's write a wrong statement:

```
(try-proof 'my-and-intro
  (assume [x A
           y B]
    (assume [C :type
             H (==> A B C)]
      (have <a> (==> C C) :by (H x)))))
;; => [:failed my-and-intro
;;       {:msg "Have step elaboration failed: synthetized term type and expected type do n
;;        :have-name <a>,
;;        :expected-type (Π [⇧ C] C),
;;        :synthetized-type (Π [⇧' B] C), :meta {:line 302, :column 7}}]
```

As we can see, LaTTe is very verbose in its error message, which is often a good thing but only after some practice with the environment. Going back to our *correct* intermediate result, we can easily produce a desired term of type `C`, because we have now our function `<a>` of type `(==> B C)`.

```
(try-proof 'my-and-intro
  (assume [x A
           y B]
    (assume [C :type
             H (==> A B C)]
      (have <a> (==> B C) :by (H x))
      (have <b> C :by (<a> y)))))
;; => [:failed my-and-intro {:msg "Proof incomplete."}]
```

You would maybe expect that the proof was complete, but what we did here was just asserting two intermediate propositions under the specified assumptions. What LaTTe did was to check these to be correct, but we are still missing the connection with our initial objective. To conclude our proof we have to use the `qed` form, as follows:

```
(try-proof 'my-and-intro
  (assume [x A
           y B]
    (assume [C :type
             H (==> A B C)]
      (have <a> (==> B C) :by (H x))
      (have <b> C :by (<a> y))))
  (qed <b>))
;; => [:qed my-and-intro]
```

Note that we first closed our assumptions, and used the intermediate step `<b>` outside the assumptions. Technically speaking, this builds up the term `<b>` within the required lambdas. Put in other terms, our direct proof and this declarative version are exactly the same internally. But as we will see, the fact that proofs can be elaborated in a step-by-step manner like what we did last really makes LaTTe a proof assistant and not just a type-checker. We can do *real* mathematics with only three constructs: `assume`, `have` and a final `qed`.

And now we can write our example:

```
(example [[A :type]]
    (==> A A (my-and A A))
  (qed (my-and-intro A A)))
;; => [:checked :example]
```

In the LaTTe prelude the introduction rule is called `and-intro-thm` thus we can rewrite the example as follows:

```
(example [[A :type]]
    (==> A A (and A A))
  (qed (p/and-intro-thm A A)))
;; => [:checked :example]
```

We'll see that there is a variant named `and-intro` in the prelude that is used much more often in practice.

Elimination rules

Let's see now if we can do the same for the elimination rules. There are two ways to "eliminate" a conjunction:

```
 (and A B)                      (and A B)
=========== (and-elim-left)    ============ (and-elim-right)
     A                              B
```

This is obvious: if both `A` and `B` hold then *either* `A` or `B` can be deduced. Let's state and prove the left version.

```
(defthm my-and-elim-left
  "Left-elimination of conjunction."
  [[A :type] [B :type]]
  (==> (my-and A B)
       A))
```

The proof for this is easy but not trivial. Let's first sketch it. Having `(and A B)` means `(forall [C :type] (==> (==> A B C) C))` by definition. Hence, specializing for proposition `A` we get `(==> (==> A B A) A)`. (we just replaced `C` by `A` in the definition). So we can obtained our expected goal `A` if we can prove that `(==> A B A)` is true. Let's state this as an intermediate lemma:

```
(deflemma my-impl-ignore
  [[A :type] [B :type]]
  (==> A B A))
```

A *lemma* is like a theorem but with the purpose of being used as an intermediate step in the proof of an *actual* theorem. The way we interprete this in Clojure is that a theorem is exported in the namespace whereas a lemma remains *private*. This can be of course changed by playing with Clojure's metadata (a very neat mechanism by the way!). I find it quite interesting to give precise interpretation of relatively subjective mathematical concepts: a theorem is public/exported, a lemma is not.

The proof of the lemma is straightforward so let's write it in the declarative style:

```
(proof 'my-impl-ignore
  (assume [x A
           y B]
    (have <a> A :by x))
  (qed <a>))
```

And now we are ready for the proof of the left elimination:

```
(proof 'my-and-elim-left
  (assume [H (my-and A B)]
    "We first specialize the definition of `my-and`"
    (have <a> (==> (==> A B A) A) :by (H A))
    "Then we use our intermediate lemma"
    (have <b> (==> A B A) :by (my-impl-ignore A B))
    "And we reach our goal"
    (have <c> A :by (<a> <b>)))
  (qed <c>))
```

Note that we intersped some comments in the proof, so that it can be read almost like a traditional mathematical proof.

In the LaTTe prelude, the left elimination rule is called `and-elim-left-thm`. There is a variant, named `and-elim-left`, that is used in most cases, we'll see why in a moment.

Exercise: state and prove the right elimination rule: `my-and-elim-right`. (in the prelude, the rule is `and-elim-right-thm` and the one used in practice is `and-elim-right`).

Now that we have our reasoning rules for conjunction, let's try to use them. For this we state the following proposition (as a side note, I personnally like to call a yet unproven theorem a proposition).

```
(defthm my-and-trans
  "A kind of transitivity for conjunction."
  [[A :type] [B :type] [C :type]]
  (==> (my-and A B) (my-and C B)
       (my-and A C)))
```

This is not a very interesting proposition, but it should hold and intuitively we should only need the introduciton and left-elimination rules we just proved.

Let's write again our proof sketch before writing the LaTTe code (in practice, we would rather use the incremental construction of "incomplete proofs" until reaching our goal, but we'll start to be less verbose from now on).

First, our assumptions are, say: - an hypothesis that `(my-and A B)` holds, let's call it `H1` - an hypothesis tha `(my-and C B)` holds, let's call it `H2` Now, we'll first prove that `A` holds, using our left elimination rule. The definition of the term `(my-and-elim-left A B)` is the following:

```
(==> (my-and A B) A)
```

Hence, in computational terms, it is a function taking as a parameter a term of type `(my-and A B)` and returning an `A`. Hence, to obtain an `A` we simply have to apply

this function to our term `H1`. Similarly, to obtain a `B` we have to apply the term `H2` to the function `(my-and C B)`. And now that we have an `A` and a `B` we can build a conjunction using the introdution rule. The complete proof is as follows:

```
(proof 'my-and-trans
  (assume [H1 (my-and A B)
           H2 (my-and C B)]
    (have <a> A :by ((my-and-elim-left A B) H1))
    (have <b> C :by ((my-and-elim-left C B) H2))
    (have <c> (my-and A C) :by ((my-and-intro A C) <a> <b>)))
  (qed <c>))
;; => [:qed my-and-trans]
```

This works! We performed our first elimnation-then-introduciton reasoning, which is a very frequent proof scheme.

But we can argue that from a mathematical point of view, the proof is a little bit verbose and with some redundancy.

LaTTe offers some extra features, being basic type theory, to address some of these issues. Let's restate our theorem based on the prelude library.

```
(defthm and-trans
  "A kind of transitivity for conjunction."
  [[A :type] [B :type] [C :type]]
  (==> (and A B) (and C B)
       (and A C)))
```

This is the same as before, but this time using the prelude definition for conjunction. Now we remark that the assumptions are written in the theorem statement, so we might wonder why we have to restate them in the proof. This is sometimes useful, because there can be some "distance" between the theorem and its proof (e.g. because we need to state and proof intermediate lemmas), but very often the proof is "just below" the theorem. Here our hypotheses are short so it's not really a problem, but sometimes hypotheses are large multiline statements. In such cases, we can simply use the underscode character `_` and let LaTTe figure out the hypotheses given the definition. Of course there is no magic: they will be exaclty like before.

So the beginning of our proof will be as follows:

```
(assume [H1 _ H2 _]
   <rest-of-the-proof>)
```

Of course we have then to look at the hypotheses in the theorem body.

In the previous proof, it was also a little bit cumbersome to explicitly write the types for the elimination and introduction rules. For example, it should be obvious for `H1` that the involved types as `A` and `B` (in this order), and similarly `C` and `B` for `H2`. In a proof assistant such as Coq or Agda, a very powerfull type inference "algorithm" (I use quotes since the problem is indecidable in type theory, i.e. the inference can fail or loops). In LaTTe I decided not to rely on such an algorithm because of its complexity: both from the implementor and the user point of view. In the current state of affairs, we use the notion of an *implicit*, which consists in allowing the user of LaTTe to write an arbitrary Clojure program to analyse and transform a term. There are many such implicits in the prelude, and in particular:

- the introduction rule for conjunction `and-intro`
- the left and right elimination rules: `and-elim-left` and `and-elim-right`.

Let's see their documentation:

```
(doc p/and-intro)
```

```
latte-prelude.prop/and-intro

(and-intro a b)
```

An introduction rule that takes a proof `a` of type `A`, a proof `b` of type `B` and yields a proof of type `(and A B)`.

This is an implicit version of [[and-intro-thm]].

```
(doc p/and-elim-left)
```

```
latte-prelude.prop/and-elim-left

(and-elim-left and-term)
```

An implicit elimination rule that takes a proof of type `(and A B)` and yields a proof of `A`.

This is an implicit version of [[and-elim-left-thm]].

```
(doc p/and-elim-right)
```

```
latte-prelude.prop/and-elim-right

(and-elim-right and-term)
```

An implicit elimination rule that takes a proof of type `(and A B)` and yields a proof of `B`.

This is an implicit version of [[and-elim-right-thm]].

Using these implicit variants, we do not need to state the types explicitly when introducing or eliminating the conjunctions. Our proof then is simplified as follows.

```
(proof 'and-trans
  (assume [H1 _ H2 _]
    (have <a> A :by (p/and-elim-left H1))
    (have <b> C :by (p/and-elim-left H2))
    (have <c> (and A C) :by (p/and-intro <a> <b>)))
  (qed <c>))
;; => [:qed and-trans]
```

N-ary variants

The conjunction is LaTTe is a binary operator, like it is always the case in mathematical logics. However as Lispers we know the interest of n-ary associative operators. While we cannot do so at the primitive level, the prelude handles such cases, as illustrated by the following examples.

```
;; n-ary introduction
(example [[A :type] [B :type] [C :type] [D :type]
          [a A] [b B] [c C] [d D]]
    (p/and* A B C D)
  (qed (p/and-intro* a b c d)))
;; => [:checked :example]

;; n-ary eliminations
(example [[A :type] [B :type] [C :type] [D :type]]
    (==> (p/and* A B C D)
         A)
  (assume [H _]
    ;; eliminate first operand (A)
    (have <a> A :by (p/and-elim* 1 H)))
  (qed <a>))
;; => [:checked :example]

(example [[A :type] [B :type] [C :type] [D :type]]
    (==> (p/and* A B C D)
         C)
  (assume [H _]
    ;; eliminate third operand (C)
    (have <a> C :by (p/and-elim* 3 H)))
  (qed <a>))
;; => [:checked :example]
```

An important advice is to avoid mixing the binary and nary constructs:

- use `and-intro`, `and-elim-left` and `and-elim-right` with binary conjunction: `and`
- use `and-intro*` and `and-elim*` with the nary variant: `and*`

Equivalence = conjunction of implications

An important construct of logic is the equivalence of two propositions, which is defined as follows in the prelude:

```
(definition <=>
  "Logical equivalence or 'if and only if'."
  [[A :type] [B :type]]
  (and (==> A B)
       (==> B A)))
```

The introduction rule is `iff-intro-thm` in the prelude, but it's quite easy to reconstruct one.

```
(defthm my-iff-intro-thm
  "Introduction of equivalence."
  [[A :type] [B :type]]
  (==> (==> A B)
       (==> B A)
```

```
       (<=> A B)))
;; => [:declared :theorem my-iff-intro-thm]

(proof 'my-iff-intro-thm
  (assume [Hif (==> A B)
           Honly-if (==> B A)]
    (have <a> (<=> A B) :by (p/and-intro Hif Honly-if)))
  (qed <a>))
;; => [:qed my-iff-intro-thm]
```

In the same spirit the elimination rules `iff-elim-if-thm` and `if-elim-only-if-thm` of the prelude are conjunction eliminations in disguise.

```
(defthm my-iff-elim-if-thm
  "Elimination of \"if\" part of an equivalence."
  [[A :type] [B :type]]
  (==> (<=> A B)
       (==> A B)))
;; => [:declared :theorem my-iff-elim-if-thm]

(proof 'my-iff-elim-if-thm
  (assume [Heq (<=> A B)]
    (have <a> (==> A B) :by (p/and-elim-left Heq)))
  (qed <a>))
;; => [:qed my-iff-elim-if-thm]
```

Exercise: define and prove the "only if" elimination.

Note that in the prelude the rules to use in practice are the implicit ones: `iff-intro`, `iff-elim-if` and `iff-elim-only-if`.

```
(example [[A :type] [B :type]]
    (==> (==> A B)
         (==> B A)
         (<=> A B))
  (assume [H1 _ H2 _]
    (have <a> (<=> A B) :by (p/iff-intro H1 H2)))
  (qed <a>))
;; => [:checked :example]

(example [[A :type] [B :type]]
    (==> (<=> A B)
         (==> A B))
  (qed (lambda [H (<=> A B)]
         (p/iff-elim-if H))))
;; => [:checked :example]

(example [[A :type] [B :type]]
    (==> (<=> A B)
         (==> B A))
  (qed (lambda [H (<=> A B)]
         (p/iff-elim-only-if H))))
```

```
;; => [:checked :example]
```

With some practice, and inspecting the quite readable sources of the prelude, you will quickly master the use of implicits. But don't forget that the non-implicit versions are always useable. Moreover, implicit variants are not always proposed, so consulting the documentation is always a good thing.

Disjunction and proof-by-cases

Instead of reconstructing things like we did with conjunction, we will directly use the introduction and elimination for disjunction (logical *or*) as they are defined in the prelude, and not redefine them (it is a good exercise).

There are two introduction rules and one elimination rule for `or`, which is exactly the opposite of conjunction. The introduction rules are as follows:

```
(defthm or-intro-left-thm
  [[A :type] [B :type]]
  (==> A
       (or A B)))

(defthm or-intro-right-thm
  [[A :type] [B :type]]
  (==> B
       (or A B)))
```

The meaning is obvious, if `A` holds then the disjunction `A` *or* `B` also holds, this is the left introduction. And the right introduction can be applied if `B` holds. We can also use the *implicits* `or-intro-left` (resp. `or-intro-right`) with which the types `B` (resp. `A`) are inferred.

There are of course implicit variants of the rules.

```
(doc p/or-intro-left)
```

```
     (or-intro-left proofA B)```

     Left introduction for a disjunction `(or A B)`, with `proofA` a proof of `A`.
```

```
(doc p/or-intro-right)
```

```
     (or-intro-right A proofB)```

     Right introduction for a disjunction `(or A B)`, with `proofB` a proof of `B`.
```

As an illustration, if you know that both `A` and `B` holds, then there are two ways to prove that either of them holds.

```
(example [[A :type] [B :type]]
    (==> (and A B)
         (or A B))
  ;; The "left" proof:
  (assume [H (and A B)]
```

```
    (have <a> A :by (p/and-elim-left H))
    "We have A thus we can introduce (or A B) from the left"
    (have <b> (or A B) :by (p/or-intro-left <a> B)))
                           ;; a.k.a. ((p/or-intro-left-thm A B) <a>)))
  (qed <b>))
;; => [:checked :example]

(example [[A :type] [B :type]]
    (==> (and A B)
         (or A B))
  ;; The "right" proof:
  (assume [H (and A B)]
    (have <a> B :by (p/and-elim-right H))
    "We have B thus we can introduce (or A B) from the right"
    (have <b> (or A B) :by (p/or-intro-right A <a>)))
                           ;; a.k.a. ((p/or-intro-right-thm A B) <a>)))
  (qed <b>))
;; => [:checked :example]
```

The elimination rule is a little bit more complex because it implements a very general *proof-by-case* scheme.

```
(defthm or-elim-thm
  [[A :type] [B :type]]
  (==> (or A B)
       (forall [C :type]
         (==> (==> A C)
              (==> B C)
              C))))
```

Its informal reading is as follows. If you know that either the proposition `A` or the proposition `B` holds (it is not exclusive: `A` and `B` may be true also, but then it is either to use this fact), then suppose your goal is to prove some proposition `C`. Then you have two things to do to demonstrate that `C` holds:

- the first "left" case: under the assumption that `A` holds, you prove that also `C` holds, i.e. `A` implies `C`
- the second "right" case: you prove that `B` implies `C` also

Hence, since in both case `C` is true you know that `(or A B)` is enough as an assumption. There is of course an implicit version `or-elim` which is the one we will use in practice.

```
(doc p/or-elim)
```

> `(or-elim or-term prop left-proof right-proof)`
>
> An elimination rule that takes a proof `or-term` of type `(or A B)`, a proposition `prop`, a proof `left-proof` of type `(==> A prop)`, a proof `right-proof` of type `(==> B prop)`, and thus concludes that `prop` holds by `or-elim-thm`.

As an illustration, let's prove the following theorem:

```
(defthm my-or-assoc
  "Associativity of disjunction"
  [[A :type] [B :type] [C :type]]
  (==> (or A (or B C))
       (or (or A B) C)))
```

This is an expected property of disjunction: associativity. Our assumption is `(or A (or B C))`, thus a disjunction and the rule `or-elim` gives the proof-by-case scheme we should follows:

- the first case is for the assumption of `A` alone,
- the second case is for assumption `(or B C)`

If in both cases we can reach the conclusion, then we'll have a proof of the theorem… Let's try this…

```
(try-proof 'my-or-assoc
  (assume [H (or A (or B C))]
    "First case, we assume `A`."
    (assume [H1 A]
      (have <a1> (or A B) :by (p/or-intro-left H1 B))
      "We reach our first goal below."
      (have <a> (or (or A B) C) :by (p/or-intro-left <a1> C)))
    "Second case, we assume `(or B C)`"
    (assume [H2 (or B C)]
      "...")))
;; => [:failed my-or-assoc {:msg "Proof incomplete."}]
```

The second case is a little more complex. Our goal is to prove the following, expressed as a lemma:

```
(deflemma my-or-assoc-aux
  [[A :type] [B :type] [C :type]]
  (==> (or B C)
       (or (or A B) C)))

(proof 'my-or-assoc-aux
  (assume [H (or B C)]
    "First case: assuming `B`"
    (assume [H1 B]
      (have <a1> (or A B) :by (p/or-intro-right A H1))
      "Ok, we reach the goal"
      (have <a> (or (or A B) C) :by (p/or-intro-left <a1> C)))
    "Seconde case: assuming `C`"
    (assume [H2 C]
      "This is immediate."
      (have <b> (or (or A B) C) :by (p/or-intro-right (or A B) H2)))
    "We can now use the or-elimination"
    (have <c> _ :by (p/or-elim H ; this is the disjunction to eliminate
                               (or (or A B) C) ; and this is ou goal
                               <a> ; proof of the first case
                               <b> ; proof of the second case
```

```
                                  )))
  "We can conclude"
  (qed <c>))
;; => [:qed my-or-assoc-aux]
```

At proof step `<c>`, what we want to "have" is `(or (or A B) C)` and we us the `or-elim` rule to obtain this. The reason we use the underscode placeholder `_` is that the type we expect is already the second argument or `or-elim` so there is nothing to gain to repeat it. The `or-elim` is one of the rare occasion when using the underscode is actually useful in `have` steps.

Now that we demonstrated our Lemma, we can finish the proof for our main theorem.

```
(proof 'my-or-assoc
  (assume [H (or A (or B C))]
    "First case, we assume `A`."
    (assume [H1 A]
      (have <a1> (or A B) :by (p/or-intro-left H1 B))
      "We reach our first goal below."
      (have <a> (or (or A B) C) :by (p/or-intro-left <a1> C)))
    "Second case, we assume `(or B C)`"
    (assume [H2 (or B C)]
      (have <b> (or (or A B) C) :by ((my-or-assoc-aux A B C) H2)))
    "We eliminate the disjunction."
    (have <c> _ :by (p/or-elim H (or (or A B) C) <a> <b>)))
  (qed <c>))
;; => [:qed my-or-assoc]
```

Tadaaa! We did it, we reused an auxiliary lemma to prove a main theorem. And we also used the elimination rule for disjunction twice-in-a-row, I think we should be satisfied somehow.

Exercise: state and prove the following:

- Under the assumption `(or A A)` prove `A` (for an arbitrary proposition `A`)
- From `(or A B)` prove `(or B A)`

Proof by "many" cases

There is also an n-ary variant `or*` for disjunction. There is only ones generic introduction rule, as illustrated in the following examples:

```
(example [[A :type] [B :type] [C :type] [D :type]]
    (==> B (or* A B C D))
  (qed (lambda [x B] (p/or-intro* A x C D))))
;; => [:checked :example]

(example [[A :type] [B :type] [C :type] [D :type]]
    (==> D (or* A B C D))
  (qed (lambda [x D] (p/or-intro* A B C x))))
;; => [:checked :example]
```

The elimination rules enables to make a proof with more than two cases, without having to nest multiple occurrences of `or-elim`.

```
(example [[A :type] [B :type] [C :type] [D :type] [E :type]]
    (==> (or* A B C D)
         (==> A E)
         (==> B E)
         (==> C E)
         (==> D E)
         E)
  (assume [Hor _ HA _ HB _ HC _ HD _]
    (have <a> _ :by (p/or-elim* Hor ;; the or* term
                                E ;; our goal
                                HA HB HC HD ;; the 4 cases
                                )))
  (qed <a>))
;; => [:checked :example]
```

Proving by contradiction

It is not because we are doing *logic* that we should *absurdity* at all cost. It is in fact definable in type theory, and in LaTTe it is as follows:

```
(definition absurd
  "Absurdity."
  []
  (forall [α :type] α))
```

If read litteraly, this definition means *everything is provable*. It is thankfully safe to *define* such a concept, what would be dramatic would be if we could prove it as a theorem. Of course, without introducing an axiom (and trusting type theory and its implementation in LaTTe), it is not possible to find a proof of `absurd`. We might wonder, then, what is the interest of writing such an "absurd" formal definition! The first element of answer is the following incursion of the venerable latin language:

```
(defthm ex-falso
 "Ex falso sequitur quodlibet
  (proof by contradiction, elimination for absurdity)."
 [[A :type]]
 (==> absurd A))
```

This theorem statement is the definition of *a proof by contradiction*. One of the most frequent "fake news" about intuitionistic logic (and constructive mathematics) is that it would not be possible to make such a proof by contradiction, everything should be "true", and classical reasoning is required to reason about "false" things. Of course this is totally incorrect. We can as an illustration demonstrate the *ex-falso* proof principle:

```
(try-example [[A :type]]
    (==> p/absurd A)
  ;; the proof of absurdity
```

```
  (assume [H p/absurd]
    (have <a> A :by (H A)))
  (qed <a>))
;; => [:checked :example]
```

The proof principle available only in classical logic is the *principle of the exluded middle*, the fact that a proposition can only be true or false. So one can reason that something is true because it "cannot" be false. In intuitionistic logic, one must show *how* something is true or false. We will go back to this discussion in a later chapter but for now just remember that *proof by contradiction* is a different matter.

In terms of reasoning, the *ex-falso* principle is important. It says that when an assumption is *contradictory*, then it is by showing that absurd is derivable from it. If we manage to do so, then the `ex-falso` theorem allows to prove the expected conclusion.

Before making some trivial examples of proofs by contradiction, the last piece of the puzzle is the operator of logical negation. In type theory it is defined from the absurdity, and in LaTTe it is the following:

```
(definition not
  "Logical negation."
  [[A :type]]
 (==> A absurd))
```

Let's put this definition in practice. One basic principle of logic is that nothing can be *at the same* true and false, thus the following should hold:

```
(defthm not-and
  [[A :type]]
  (not (and A (not A))))
```

Note that the statement can be rewritten `(==> (and A (not A)) p/absurd)`.

```
(proof 'not-and
  "We first restate the hypothesis."
  (assume [H (and A (not A))]
    "First, we have `A` by assumption."
    (have <a> A :by (p/and-elim-left H))
    "Now, we can 'feed' it to `(not A)`"
    (have <b> p/absurd :by ((p/and-elim-right H) <a>)))
  "this is absurd!"
  (qed <b>))
;; => [:qed not-and]
```

We will see other proofs by contradiction in the rest of the tutorial.

Exercise: prove the following theorem about "double negation".

```
(defthm my-impl-not-not
  [[A :type]]
  (==> A (not (not A))))
```

What about the converse? Can you prove it?

To be or not to be? That is the existential question!

There is no direct support for *existential quantification* in the kernel theory of LaTTe, unlike the universal (∀) case. However, there is a beautiful encoding of the existential quantifier, which is defined in the quant library of the LaTTe prelude. It is as follows.

```
(definition ex-def
  [[T :type] [P (==> T :type)]]
 (forall [α :type]
   (==> (forall [x T] (==> (P x) α))
        α)))
```

The details are not so important but roughly if we write in LaTTe:

```
(ex-def T (λ [x T] (P x)))
```

for all intent and purpose it has the same logical meaning as saying:

> there exists an `x` of type `T` such that `(P x)` is true.

In LaTTe a convenient *notation* is defined so that `ex-def` expressions are written in a more conventional manner, as follows:

```
(exists [x T] (P x))
```

Thus, we now how to write an existential quantification, but *what is* precisely the logical meaning of it requires the associated introduction and elimination rules.

Introduction rule

For the introduction rule we have:

```
(defthm ex-intro-thm
  [[T :type] [P (==> T :type)] [x T]]
  (==> (P x)
       (ex P)))
```

The notation `(ex P)` is a shortcut for when we do not need to isolated the `x` variable in the existential (and `(ex P)` is exactly the same as `(ex-def T P)` with `T` the domain of `P` that is in fact inferred from it). So the rule simply says that if the predicate `P` is true for some `x` then the existential quantification old.

> If `x` is a jupitarian, then there *exists* a jupitarian!

Exercise: Given the following definition

```
(definition my-ex-def
  [[T :type] [P (==> T :type)]]
  (forall [A :type]
     (==> (forall [x T] (==> (P x) A))
          A)))
```

Prove the following:

```
(defthm my-ex-intro-thm
  [[T :type] [P (==> T :type)] [x T]]
```

```
  (==> (P x)
       (my-ex-def T P)))
```

Hint: look at the definition of `my-ex-def`, it's just forall and implication, so it just a question of following the rules of the game presented in the previous chapter.

Here is an example of usage of the introduction rule.

```
(defthm ex-and-intro
  [[T :type] [P (==> T :type)] [Q (==> T :type)] [x T]]
  (==> (P x)
       (Q x)
       (exists [y T] (and (P y) (Q y)))))
```

The proof is rather simple.

```
(proof 'ex-and-intro
  (assume [HP _ HQ _]
    (have <a> (and (P x) (Q x)) :by (p/and-intro HP HQ))
    (have <b> (exists [y T] (and (P y) (Q y)))
          :by ((q/ex-intro (lambda [y T] (and (P y) (Q y))) x) <a>)))
  (qed <b>))
;; => [:qed ex-and-intro]
```

The only non-trival part is the `<b>` step. We are invoking the `ex-intro` rule which require a predicate `P` and a witness `x` that the predicate holds. We then provide the result of step `<a>`, i.e. a proof that `(P x)` holds to the introduction rule, which produces the required existential.

Exercise: redo the last proof using your own `my-ex-intro-thm`.

Elimination rule

The elimination rule for the existential is the following one:

```
(defthm ex-elim-thm
  [[T :type] [P (==> T :type)] [A :type]]
  (==> (ex P)
       (forall [x T] (==> (P x) A))
       A))
```

Considering an arbitrary proposition `A`, the definition says that if we know that there exists an `x` of type `T` such that `(P x)` (which is simply denoted by `(ex P)`), and moreover if forall `x`'s of type `T` from `(P x)` we can deduce `A`, then `A` is true.

One way to explain this definition is that the assumption `(forall [x T] (==> (P x) A))` is very strong. We could deduce `A` by exhibiting a `z` such that `(P z)` and we would obtain `A`. But since we know that one exists, we don't need to explicitely provide this particular `z`. Let's prove this manually, using directly `my-ex-def`.

```
(defthm my-ex-elim-thm
  [[T :type] [P (==> T :type)] [A :type]]
  (==> (my-ex-def T P)
       (forall [x T] (==> (P x) A))
```

```
       A))

(proof 'my-ex-elim-thm
  (assume [Hex (my-ex-def T P)
           Hall (forall [x T] (==> (P x) A))]
    "First let's expand the definition of `my-ex-def`."
    (have <a> (==> (forall [x T] (==> (P x) A))
                   A) :by (Hex A))
    "And know it is easy to obtain `A`."
    (have <b> A :by (<a> Hall)))
  "And we're done."
  (qed <b>))
;; => [:qed my-ex-elim-thm]
```

Let's consider a first example.

```
(defthm ex-and-elim-left
  [[T :type] [P (==> T :type)] [Q (==> T :type)]]
  (==> (exists [x T] (and (P x) (Q x)))
       (exists [x T] (P x))))
```

One possible proof is as follows.

```
(proof 'ex-and-elim
  (assume [Hex _]
    (assume [y T
             Hy (and (P y) (Q y))]
      (have <a> (P y) :by (p/and-elim-left Hy))
      "We need to introduce the existential of the goal."
      (have <b> (exists [x T] (P x))
            :by ((q/ex-intro (lambda [x T] (P x)) y) <a>)))
    "Now we are ready for the existential elimination"
    (have <c> _  ;; the obtain proposition is the second argument
          :by ((q/ex-elim (lambda [x T] (and (P x) (Q x)))
                          ;; below is what we want
                          (exists [x T] (P x)))
               Hex ;; the proof for the existential
               <b> ;; the proof for the generalization
               )))
  (qed <c>))
;; => [:qed ex-and-elim]
```

In this proof we used the `ex-elim` implicit, which is like `ex-elim-thm` but with the first type synthetized, i.e. `(ex-elim P x)` is the same as `(ex-elim-thm T P x)` provided `P` is of type `(==> T :type)`.

Note that when applying the elimination rule (step `<c>` above), we used the underscore `_` so that the type of the step is inferred. This is because the type we target is an argument of the elimination rule. In fact the situation is similar to the `or-elim` rule for disjunction, and indeed there is a tight connection between existential and or. Consider the following statement:

> a prime number is 2, or 3, or, 5, or etc. (infinitely)

This does not define the prime numbers, but merely says that we could list them all because there is *at least* one of them. So from a purely logical point of view, the previous informal statement (an infinite disjunction) can be formalized as a single existential:

there exists some prime number

Exercise : state and prove the complement right elimination. Try to use your own `my-ex-elim-thm` in the proof.

We will now consider a more complex example of a proof involving the existential. The statemenent is as follows:

```
(defthm exists-commute
  [[A :type] [B :type] [P (==> A B :type)]]
  (==> (exists [a A]
         (exists [b B] (P a b)))
       (exists [b B]
         (exists [a A] (P a b)))))
```

This is an apparently trivial property of nested existentials: the order of the quantifier does not matter. While this sounds like a simple property, we already know that we will have to eliminate two existentials in the hypothesis, and we will then have to introduce two existentials in the conclusion. Hence, we will need to use both `q/ex-intro` and `q/ex-elim` twice, but in a nested way. So we can alreay imagine that the proof structure is not as simple as it may seem.

To simplify a little bit the matter, we will also introduce a local definition using the `pose` proof command. Writing:

```
(pose X := <expr>)
```

is exactly the same as writing:

```
(have X _ :by <expr>)
```

Let's dig into the commutation proof.

```
(proof 'exists-commute
  (assume [Hex _] ;; cf. the assumption in the theorem
    "We state our goal so that we do not have to write it everywhere."
    (pose GOAL := (exists [b B]
                    (exists [a A] (P a b))))
    "We begin to eliminate the outer existential (for variable `a`)"
    (assume [x A
             Hx (exists [b B] (P x b))]
      "The hypothesis `Hx` is our (P x) in the elimination rule.
our objective is to prove the `GOAL` from it, but first we have
to eliminate the inner existential (for variable `b`)."
      (assume [y B
               Hy (P x y)]
        "We introduce first the inner existential of the `GOAL`
(i.e. for variable `a`)."
        (have <a> (exists [a A] (P a y))
              :by ((q/ex-intro (lambda [a A] (P a y)) x) Hy))
```

```
        "And the we introduce the outer existential, reaching the `GOAL`."
        (have <b> GOAL
              :by ((q/ex-intro (lambda [b B] (exists [a A] (P a b))) y)
                   <a>)))
      "We are now ready to eliminate the inner existential."
      (have <c> _ :by ((q/ex-elim (lambda [y B] (P x y)) GOAL)
                       Hx <b>)))
    "And we can ultimately eliminate the outer existential."
    (have <d> _ :by ((q/ex-elim (lambda [x A] (exists [b B] (P x b))) GOAL)
                     Hex <c>)))
  "We're done!"
  (qed <d>))
;; => [:qed exists-commute]
```

This should illustrate the relative complexity of the existential proof methods, especially the elimination part. However, we should keep in mind that existential reasoning is quite a powerful concept! I do not think there's too much incidental complexity involved here.

Exercise : prove the following theorem.

```
(defthm ex-impl
  [[T :type] [P (==> T :type)] [Q (==> T :type)]]
  (==> (exists [x T] (P x))
       (forall [y T] (==> (P y) (Q y)))
       (exists [x T] (Q x))))
```

Natural logic, in summary

In this chapter, we discussed the following topics.

- Natural logic is based on introduction and elimination rules,
- The implication and universal quantification are primitive in type theory in LaTTe. They are introduced with `lambda` and eliminated with function application.
- Conjunction (logical `and`) is a derived principles, with rules `and-intro`, `and-elim-left` and `and-elim-righ` (also the n-ary variants `and*` etc.).
- Logical equivalence is a conjunction of two implications, thus ultimately a conjunction.
- Disjunction (logical `or`) is also a derived principle. The rules are `or-intro-left`, `or-intro-right` and `or-elim`. The elimination rule implements the important principle of *proof by case*. For more than two cases, you may use `or*`, `or-intro*` and `or-elim*`.
- Negation (logical `not`) is derived from absurdity, and from absurdity one can prove everything. This enables *proofs by contradiction* in LaTTe.
- There is a simple yet effective encoding of existential quantification in type theory, with deduction rules `ex-intro` and `ex-elim`. ;; Now that we have a fairly complete logic, we will start to see how to do more traditional mathematics in LaTTe, starting with *set theory*.

A bag full of sets

For both trained mathematicians and laypersons, set theory is at the root of mathematics. And indeed, most mathematical objects can be founded upon the axiomatic set theory, namely ZF or ZFC. But sets are also used almost everywhere in maths and, well, everywhere. Even children practice sets quite early: think "the set of available ice cream flavors"!

Type theory (TT) is, in a way, an alternative construction of mathematical objects, and while TT can be justified in terms of "standard" set theory (through the so-called BHK-interpretation), it clearly has a foundational role. Hence if the "foundational" sets and "layperson" sets are in general not distinguished, type theory makes such a distinction in that types replace sets in the foundational role. But what about the "layperson" sets? Can't children select their prefered ice cream flavour in type theory?

```
(ns latte-tutorial.ch05-set-theory
  (:require [latte.core :as latte :refer [definition defthm deflemma
                                          lambda forall
                                          example try-example
                                          proof try-proof
                                          assume have qed]]
            [latte-prelude.prop :as p :refer [and and* or or* not <=>]]
            [clojure.repl :refer [doc]]

            ;; set theory requirements
            [latte-sets.core :as set :refer [set elem set-of subset seteq]]
            ;; boolean algebra of sets
            [latte-sets.algebra :as setops :refer [union inter diff complement]]
            )


  ;; also, these belong to logic or formal arithmetic
  (:refer-clojure :exclude [and or not set complement]))
```

As a first approach, it seems natural to think of types as sets. In typed programming saying that "a variable is of type `int`" is the same as saying that it can store a value in the set of (machine-representable) integers. In the library `latte-integers` there is such a type, and indeed all the (infinite) integer numbers inhabit this type. But suppose now we would like to talk about the set of positive integer. In mathematics we would use the following notation:

$$\{n \in \mathbb{Z} \mid n \geq 0\}$$

In set theory we would also probably say that this is the set $\mathbb{N}$ (although sometimes the zero is omitted, which it clearly shouldn't!). It is possible to define a type `nat` for the natural number (e.g. as a variant of the way `int` is constructed), however it would not be a subset of `int`.

What we are looking for is a way to constrains a type, i.e. something of the form:

$$\{x : T \mid P(x)\}$$

The set of all x's of type T such that $P(x)$ is true, for some predicate P. Some proof assistants introduce the notion of a "sigma-type" (Σ-type) to enable the construction of such a "subset of a type". But in fact, we have already all the necessary tools in LaTTe to make such a construction.

If we consider that `P` is of type `(==> T :type)` then we can build our set as follows:

```
(lambda [x T] (P x))
```

This function represents exactly what we need: all the `x`'s of type `T` such that `(P x)` holds. We will now see how much (if not all?) of the "layperson" set theory can be rebuilt based on this simple idea.

As a first example, we can define the "emptyset" of type `T`.

```
(definition my-emptyset [[T :type]]
  (lambda [x T] p/absurd))
;; => [:defined :term my-emptyset]
```

We know from the previous chapter that there cannot be any value of type `absurd`, thus there is no `x` to satisfy `my-emptyset`, and thus `my-emptyset` is empty! Remark that unlike in classical set theory, there is no universal notion of an emptyset in type theory. Each type `T` possesses its own emptyset: this is *typed* set theory!

Set membership

The definition of a set (precisely `latte-sets.core/set`) in LaTTe is as follows:

```
(definition set
  "The type of sets whose elements are of type `T`."
  [[T :type]]
  (==> T :type))
```

So it is a function from a type `T` to `:type`, the type of propositions. Put in other terms, it is a proposition conditioned by a type. We can check that our definition of an emptyset satisfies this definition.

```
(latte/type-of [T :type]
               (my-emptyset T))
;; => (==> T *)
```

Set theory is all about *membership*: making a clear separation about who is in the type, and who isn't? In LaTTe set membership is very simple: it is (again!) function application.

For an element `e` and a set `E`, `e∈E` iff `(E e)` !

As an example, we can show that an emptyset cannot contain any element.

```
(defthm my-emptyset-empty [[T :type]]
  (forall [x T] (not ((my-emptyset T) x))))
;; => [:declared :theorem my-emptyset-empty]
```

The proof is by contradiction, and it is very simple.

```
(proof 'my-emptyset-empty
  "We assume that x is in the emptyset ..."
  (assume [x T
           H ((my-emptyset T) x)]
    "... and it is absurd by definition of the emptyset"
    (have <a> p/absurd :by H))
  (qed <a>))
;; => [:qed my-emptyset-empty]
```

The only drawback of the "lambda-as-set" approach is that the way things are written do not look like the traditional set notations. The `latte-sets` library introduce the notation `(elem e E)` which looks more like the traditional notation `e∈E` than `(E e)`. Thus, we can rewrite our theorem and proof as follows:

```
(defthm my-emptyset-empty-v2 [[T :type]]
  (forall [x T] (not (elem x (my-emptyset T)))))
;; => [:declared :theorem my-emptyset-empty-v2]

(proof 'my-emptyset-empty-v2
  (assume [x T
           H (elem x (my-emptyset T))]
    (have <a> p/absurd :by H))
  (qed <a>))
;; => [:qed my-emptyset-empty-v2]
```

This is probably more readable for the mathematics (and lisp) practionner. Note, also, that there is aspecific notation for the definition of a set by comprehension:

```
{ x : T | P(x) }  is  (set-of [x T] (P x))
... which is (lambda [x T] (P x))
```

It is important to remember that `set-of` constructions are still `lambda`'s.

Subsets and set-equality

There is an important relationship between implication and being *a subset of* another set. This is clearly apparent in the following definition:

```
(definition subset-def
  "The subset relation for type `T`.
The expression `(subset-def T s1 s2)` means that
 the set `s1` is a subset of `s2`, i.e. `s1`⊆`s2`."
  [[T :type] [s1 (set T)] [s2 (set T)]]
```

```
  (forall [x T]
    (==> (elem x s1)
         (elem x s2))))
```

In the library a shortcut `(subset s1 s2)` is also defined. The definition says that a set (`s2`) is a subset of another set (`s1`), both defined over a type `T`, if for any element `x` of the considered type `T`, `x∈s1` implies `x∈s2`. If we rewrite slightly our two sets as follows:

```
s1 ≡ { x:T | P1(x) }  and s1 ≡ { y:T | P2(x) }
```

The definition simply says that `P1` should imply `P2`. You probably know that the emptyset is vacuously a subset of any other set. Let's prove this as a Lemma.

```
(deflemma empty-subset
  [[T :type] [s (set T)]]
  (subset (my-emptyset T) s))

(proof 'empty-subset
  (assume [x T
           Hempty (elem x (my-emptyset T))]
    "Of course we proceed by contradiction,
because we know that there is no such element x"
    (have <a> (not (elem x (my-emptyset T)))
          :by ((my-emptyset-empty T) x))
    "Hence we obtain absurdity"
    (have <b> p/absurd :by (<a> Hempty))
    "We can have anything we want here!"
    (have <c> (elem x s) :by (<b> (elem x s))))
  (qed <c>))
;; => [:qed empty-subset]
```

Exercise: prove the following theorems:

- `subset-refl`: the subset relation is *reflexive*: `(subset s s)` for any set `s`
- `subset-trans`: it is *transitive*: if `(subset s1 s2)` and `(subset s2 s3)` then `(subset s1 s3)`

We can define a useful notion of equality based on the subset relation. It is technically-speaking rather an equivalence relation, so let's call it *"set equivalence"*.

```
(definition seteq-def
  "Set equivalence.
  This is a natural notion of \"equal sets\"
  based on the subset relation."
  [[T :type] [s1 (set T)] [s2 (set T)]]
  (and (subset s1 s2)
       (subset s2 s1)))
```

And you may rewrite `(seteq-def T s1 s2)` as `(seteq s1 s2)`.

It is rather a trivial fact that this equivalence is symmetric.

```
(defthm seteq-sym
  [[T :type] [s1 (set T)] [s2 (set T)]]
```

```
  (==> (seteq s1 s2)
       (seteq s2 s1)))
```

The proof is a one-liner.

```
(proof 'seteq-sym
  (qed (lambda [H (seteq s1 s2)] (p/and-sym H))))
;; => [:qed seteq-sym]
```

Well, maybe it's a little bit too crytpic, let's decompose the proof.

```
(proof 'seteq-sym
  (assume [H (seteq s1 s2)]
    "We can decompose the conjunction."
    (have <a> (subset s1 s2) :by (p/and-elim-left H))
    (have <b> (subset s2 s1) :by (p/and-elim-right H))
    "Now let's rebuild a conjunction by reversing
the two previous steps"
    (have <c> (and (subset s2 s1)
                   (subset s1 s2))
          :by (p/and-intro <b> <a>))
    "This is set equivalence!"
    (have <d> (seteq s2 s1) :by <c>))
  (qed <d>))
  ;;=>  [:qed seteq-sym]
```

The following exercise (if solved) shows that `seteq` is a proper *equivalence relation* (together with symmetry).

Exercise: prove the following theorems:

- `seteq-refl`: reflexivity of set equality
- `seteq-trans`: transitivity of set equality‘

We will go back to this notion of equality/equivalence, and compare it with other "competing" notions.

Boolean algebra of sets

One major contribution of *George Boole* was its discovery (and study) of the relation between algebra (at that time, mostly developed for numbers) and logic. There is for example a simple correspondance between calculating in base 2 arithmetics, and propositional logic. `True` is 1, `False` is 0, `and` is `times`, `or` is `plus`, etc.

Important algebraic properties follow the correspondance: `False` is a unit of disjunction and "absorbing" for conjunection, `True` is simply the converse.

A similar boolean algebra of sets can be constructed. More precisely, for each type `T` a specific boolean algebra can be constructed, based on the following ingredients:

- the emptyset corresponds to `False`
- the fullset corresponds to `True`
- the "disjunction" ("plus") is *set union*

- the "conjunction" ("times") is *set intersection*
- the "negation" is *set complement*

Let's define (or see the definition of) all these ingredients. We already have the emptyset, so let's define the fullset.

```
(definition my-fullset
  [[T :type]]
  (set-of [x T] (not p/absurd)))
;; => [:defined :term my-fullset]
```

Everything is "not absurd", hence this should be a good candidate for being the set of "all possible `T`'s". Let's check this.

```
(deflemma my-fullset-elem
  [[T :type]]
  (forall [x T] (elem x (my-fullset T))))
;; => [:declared :lemma my-fullset-elem]

(proof 'my-fullset-elem
  (assume [x T]
    "I would like to have `(==> p/absurd p/absurd)`."
    (assume [H p/absurd]
      (have <a> p/absurd :by H))
    "And here it is!"
    (have <b> (==> p/absurd p/absurd) :by <a>)
    "This is a synonym for the following:"
    (have <c> (not p/absurd) :by <b>))
  "And we're done."
  (qed <c>))
```

This can of course be rewritten as a one-liner.

```
(proof 'my-fullset-elem
  (qed (lambda [x T]
         (lambda [H p/absurd] H))))
;; => [:qed my-fullset-elem]
```

Union

The union of two sets is a concept similar to logical disjunction. Indeed, an element is member of the union of `s1` and `s2` if it is a member of `s1` *or* a member of `s2` (or both).

This is trivial to translate this in LaTTe:

```
(definition my-union
  [[T :type] [s1 (set T)] [s2 (set T)]]
  (set-of [x T] (or (elem x s1)
                    (elem x s2))))
;; => [:defined :term my-union]
```

Let's prove, as an example, that the emptyset set is a left unit of union (like `False`

is a left unit of disjunction).

```
(defthm my-union-left-unit
  [[T :type] [s (set T)]]
  (seteq (my-union T (my-emptyset T) s)
         s))
;; => [:declared :theorem my-union-left-unit]
```

Proving `(seteq s1 s2)` consists in building a conjunction of two "sub-proofs":

- an "if" proof of `(subset s1 s2)`
- a "only-if" proof of `(subset s2 s1)`

We will state and prove each one of these as an auxiliary lemma.

```
(deflemma my-union-left-unit-if
  [[T :type] [s (set T)]]
  (subset (my-union T (my-emptyset T) s)
          s))
;; => [:declared :lemma my-union-left-unit-if]

(proof 'my-union-left-unit-if
  (assume [x T
           Hx (elem x (my-union T (my-emptyset T) s))]
    "We simplify a little bit the assumption."
    (have <Hx>  (or (elem x (my-emptyset T))
                    (elem x s)) :by Hx)
    "We have now to prove that x ∈ s.
Union is disjunction hence we consider two cases."
    (assume [H1 (elem x (my-emptyset T))]
      (have <a1> p/absurd :by ((my-emptyset-empty T) x H1))
      (have <a> (elem x s) :by (<a1> (elem x s))))
    "Second case is trivial."
    (assume [H2 (elem x s)]
      (have <b> (elem x s) :by H2))
    "Or elimination can be performed now."
    (have <c> _ :by (p/or-elim Hx (elem x s) <a> <b>)))
  (qed <c>))
;; => [:qed my-union-left-unit-if]
```

And now for the opposite "only-if" direction.

```
(deflemma my-union-left-unit-only-if
  [[T :type] [s (set T)]]
  (subset s
          (my-union T (my-emptyset T) s)))
;; => [:declared :lemma my-union-left-unit-only-if]
```

The proof is much simpler for this case.

```
(proof 'my-union-left-unit-only-if
  (assume [x T
           Hx (elem x s)]
    (have <a> (or (elem x (my-emptyset T)) (elem x s))
```

```
          :by (p/or-intro-right (elem x (my-emptyset T)) Hx)))
  (qed <a>))
;; => [:qed my-union-left-unit-only-if]
```

And now we can prove the "equivalence" theorem.

```
(proof 'my-union-left-unit
  (qed (p/and-intro (my-union-left-unit-if T s)
                    (my-union-left-unit-only-if T s))))
;; => [:qed my-union-left-unit]
```

Exercise: prove the "right unit" theorem for union.

```
(defthm my-union-right-unit
  [[T :type] [s (set T)]]
  (seteq (my-union T s (my-emptyset T))
         s))
```

Exercise: prove that the fullset is "lef-absorbing", i.e:

```
(seteq (my-union T (my-fullset T) s)
       (my-fullset T))
```

(of course it is also "right-absorbing")

Intersection

Set intersection is almost copy-and-paste for `union`, replacing `or` by `and`.

```
(definition my-inter
  [[T :type] [s1 (set T)] [s2 (set T)]]
  (set-of [x T] (and (elem x s1)
                     (elem x s2))))
;; => [:defined :term my-inter]
```

And of course, the handling of intersection mostly relies on the proof techniques developed for conjunction. And the properties are also very close (in the same spirit as union vs. disjunction).

```
(defthm my-inter-sym [[T :type] [s1 (set T)] [s2 (set T)]]
  (seteq (my-inter T s1 s2)
         (my-inter T s2 s1)))
;; => [:declared :theorem my-inter-sym]

(proof 'my-inter-sym
  "First part ==> subset"
  (assume [x T
           Hx (elem x (my-inter T s1 s2))]
    (have <a1> (elem x s1) :by (p/and-elim-left Hx))
    (have <a2> (elem x s2) :by (p/and-elim-right Hx))
    "We just have to reverse the conjunction"
    (have <a> (elem x (my-inter T s2 s1))
          :by (p/and-intro <a2> <a1>)))
```

```
  "And now the symmetric way"
  (assume [x T
           Hx (elem x (my-inter T s2 s1))]
    (have <b1> (elem x s2) :by (p/and-elim-left Hx))
    (have <b2> (elem x s1) :by (p/and-elim-right Hx))
    (have <b> (elem x (my-inter T s1 s2))
          :by (p/and-intro <b2> <b1>)))
  "Conclusion"
  (qed (p/and-intro <a> <b>)))
;; => [:qed my-inter-sym]
```

In the standard library, the intersection is written `(inter s1 s2)`.

Exercise: state and prove that: ;; - the emptyset is "absorbing" for intersection - the fullset is a unit for intersection

Complement

Logical negation is connected to the notion of a *set complement*.

```
(definition my-complement
  [[T :type] [s (set T)]]
  (set-of [x T] (not (elem x s))))
;; => [:defined :term my-complement]
```

I really like this definition, because it gives a very concise and concrete argument in favor of a "typed" reconstruction of set theory. In classical axiomatic set theory, the complement has a rather, let's say unwieldy definition. Let's e.g. what Wikipedia is saying:

> If A is a set, then the absolute complement of A (or simply the complement of A) is the set of elements not in A. In other words, if U is the universe that contains all the elements under study, and there is no need to mention it because it is obvious and unique, then the absolute complement of A is the relative complement of A in U.

from Wikipedia (retrieved Feb. 10, 2019)

In "everyday mathematics", this notion of an *universe* is not really problematic (because "it is obvious and unique"), but in formal mathematics it is a rather unsettling notion. In type theory, things are much more natural, the universe is simply the type `T` in `(set T)`. The complement is all those `T` that are not in the set `s`, a very natural definition is you ask me. ;; Now we can state and prove a natural connection between the emptyset and the fullset.

```
(defthm empty-complement [[T :type]]
  (seteq (my-complement T (my-emptyset T))
         (my-fullset T)))
;; => [:declared :theorem empty-complement]

(proof 'empty-complement
  "first part"
```

```
  (assume [x T
           Hx (elem x (my-complement T (my-emptyset T)))]
    "Since all elements are member of the fullset, we don't
even have to look at the hypothesis `Hx`"
    (have <a> (elem x (my-fullset T)) :by ((my-fullset-elem T) x)))
  "second part"
  (assume [x T
           Hx (elem x (my-fullset T))]
    "We exploit the following."
    (have <b> (not (elem x (my-emptyset T)))
          :by ((my-emptyset-empty T) x)))
  "conclusion"
  (qed (p/and-intro <a> <b>)))
;; => [:qed empty-complement]
```

In the standard library, the complement of a set `s` is `(complement s)`.

Exercise: prove the following

```
(defthm full-complement [[T :type]]
  (seteq (my-complement T (my-fullset T))
         (my-emptyset T)))
;; => [:declared :theorem full-complement]
```

Summary

In this chapter, we saw how the *primitive* notion of a set in "traditional" mathematics could be "reinvented" in type theory, but this time as a *derived* notion… from the more primitive notion of a type. ;; Also: ;; - union is simply a disjunction under a lambda - intersection is similar but for conjunction - the notion of complement is more natural in type theory ;;

There are much more interesting things to discover in the `latte-sets` standard library, especially the *relational algebra* and also *partial functions*. In this tutorial, we will go back to sets to discuss again the notion of equality, and also when talking about quantifiers.