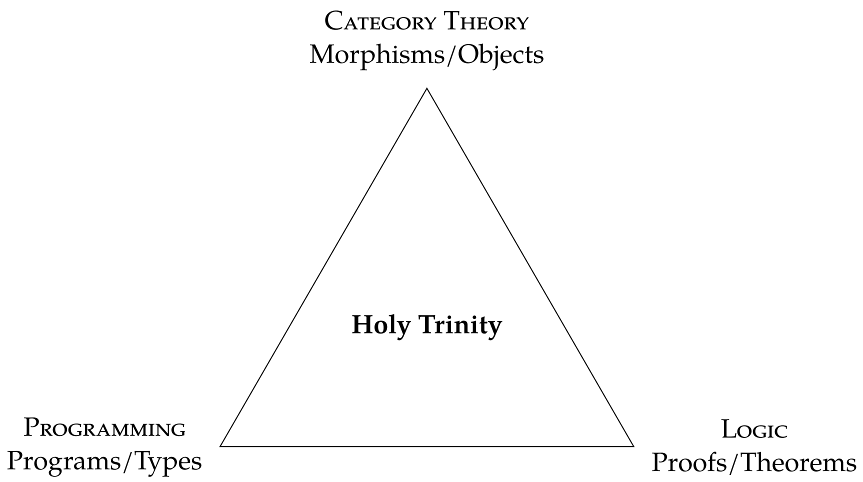
Proofs and Programs

Andrew Clifton

April 17, 2024

We're interested in "doing" logic on a computer, in the vague sense of using a program to aid in writing (correct!) proofs of theorems. The goal is to make writing proofs fun and interesting, like programming is, and not boring and tedious, as paper proofs often are. Eventually we'll see that a proof is just a special kind of program. In fact, there is a kind of "holy trinity" of concepts:



Given a logic, we can ask what it would look like as a programming language; given a programming language, we can ask what kind of logic it implicitly describes. Given a category (more specifically, a topos) we can ask what logic it describes; given a logic or programming language, we can ask what its category looks like.

1 “Doing” “logic” on a computer

We are interested in “doing” *logic* on a computer. The first questions we need to answer are

- What do we mean by *doing*?
- Which *logic* do we want to do?

For the first question I’m assuming that we want to use a computer to help with writing proofs. That is, we have some *theorem* we want to prove, and we want to use the computer to aid in that process. If we think of this as writing a function `prove` there are two possibilities:

Automated theorem prover	*Proof checker*
`prove(`*`theorem`*`)` → `proof`	`prove(`*`theorem`*`, `*`proof`*`)` → `bool`

- If we consider ourselves as writing a function which receives a *theorem* as its argument and returns a *proof object*, then we are in the realm of *automated theorem proving*, where we the function constructs the entire proof, given only the (unproved) theorem. This is the holy grail, but is not generally possible except for the simplest, most restrictive logics and theorems.
- If we consider ourselves as writing a function which receives both a *theorem* and a (potential) *proof* of it, and returns True if the proof is a valid proof of the theorem, then we are in the realm of *proof checking*. Any reasonable logic should have a mechanical procedure for *checking* proofs in it for validity, hence we only need to implement this procedure in our function.

A real-world proof assistant usually does both of these things; for small, simple theorems using a subset of the full logic, the proof assistant may be able to construct the proof automatically. Hence, one starts with a large, complex theorem and manually constructs the proof steps; but when the sub-theorems become small/simple enough, the automated theorem prover takes over and proves them for you.

In either case, we need to consider *proofs as data structure*; i.e., what do the *values* of proof objects look like? Generally speaking, the forms which proof objects can take depend on the logical connectives allowed in theorems, so we first have to consider what kind of logic we want to support.

2 First-order intuitionistic propositional logic

I'm assuming we want something fairly standard, what would be called *first-order propositional logic* (which we'll later extend to *predicate* logic). This means we have the following elements:

$A, B, C, \ldots$	(Primitive propositions)
$\top$	(True)
$\bot$	(False)
$P \wedge Q$	(And (conjunction))
$P \vee Q$	(Or (disjunction))
$P \rightarrow Q$	(Implication)

We'll assume the usual precedence rules: $\wedge$ has higher precedence than $\vee$, which has higher precedence than $\rightarrow$.

For reasons that will be explained later, we do not consider $\neg$ (negation) to be a primitive element of our logic. Instead, we define $\neg$ in terms of $\rightarrow$ and $\bot$:

$$\neg P \quad \equiv \quad P \rightarrow \bot$$

(I.e., P is not true if, from P, we can derive a contradiction.)

Similarly, we don't specifically have $\leftrightarrow$ (if-and-only-if) as a construct, but we can easily build it from what we do have:

$$A \leftrightarrow B \quad \equiv \quad (A \rightarrow B) \wedge (B \rightarrow A)$$

Although we'll give all of the above the usual definitions, we want to precisely describe *how* our logic works; we'll give this definition in the form of a collection of *inference rules* using *sequent calculus*.

An *inference rule* is written

$$\frac{\mathbf{P_1} \quad \mathbf{P_2} \quad \cdots \quad \mathbf{P_n}}{\mathbf{Q}}\;\text{Rule Name}$$

This rule can be read as

- *If all* the $\mathbf{P_i}$ are true, *then* $\mathbf{Q}$ is true, or

- To *prove* that $\mathbf{Q}$ is true, prove that all the $\mathbf{P_i}$ are true

All the $\mathbf{P_i}$ and $\mathbf{Q}$ will be *sequents*.

A *sequent* is written

$$A, B, C, \ldots \vdash X, Y, Z, \ldots$$

A sequent describes a *hypothetical judgment*:

- If, hypothetically, *all* of $A, B, C, \ldots$ (the *antecedants*) were true

- Then *at least one of* the $X, Y, Z, \ldots$ (the *consequents*) would be true.

Note that in a sequent, the elements of the left-hand-side are joined implicitly by an "And", while the elements of the right-hand-side are implicitly joined by an "Or". But the syntactical elements of a sequent (comma, $\vdash$) are just that, syntactical; they are not *operators*, you cannot nest sequents, etc.

We'll restrict our logic to *intuitionistic sequents*, where the right-hand side consists of exactly one element: $A, B, C, \ldots \vdash Q$. We'll call the list on the left-hand-side the *context*.

Any sequent calculus normally starts with *structural rules*, which describe how the lists on the left/right hand sides of the $\vdash$ behave. I'll skip over these rules, except to mention that 1) the order of the elements in the context doesn't matter, 2) duplicates are allowed (and you can freely duplicate any existing element), and 3) unused elements of the context can be dropped at will.[1] These basically correspond to our intuition that the context is a list of *facts* which we have learned or been given in the course of a proof: the order of facts doesn't matter, knowing a fact more than once is the same as knowing it once, and if you reach the end of a proof knowing "too much" (you have some facts you didn't end up using), that's not a problem.

One vital rule is the *assummption rule*, which states

$$\frac{}{\Gamma, A \vdash A}\;\text{Assume}$$

[1]Removing any of the structure rules yields a sub-structural logic; removing (1) yields a *residuated logic*, while removing (2) and/or (3) yields various other kinds of sub-structural logic such as linear logic.

(Here, and later, we use Γ to stand in for "everything else in the context".) The assumption rule states that if we assume A is true, then from this we can conclude that A is true. Generally, a proof will start with the context empty; later rules will add assumptions to the context, which can then be used by the assumption rule.

Most rules for our logical structures will come in pairs: a *left rule* which describes what we can do with the structure when it is an *assumption* (what can we do if we "have" it) and a *right* rule which describes what we have to do to *prove* the given structure (what we have to do to "get" it).

Conjunction: $\land$

Let's consider what the left and right rules should be for $\land$ (And):

- If we *have* an And as an assumption, then we should be able to use both its conjuncts freely. I.e., assuming that $A \land B$ is true should be the same as assuming that A and B are true, separately. So the $\land$-Left rule is

  $$\text{$\land$-Left}\,\frac{\Gamma, A, B \vdash Q}{\Gamma, A \land B \vdash Q}$$

  Remember that the left-hand-side of the $\vdash$ is implicitly an And, so a $\land$ on the left can simply be split up into its parts, making them available.

- If we *want* to prove an And, it seems reasonable to require that we prove both halves of it. So the Right rule for $\land$ is

  $$\text{$\land$-Right}\,\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \land Q}$$

  I.e., to show that $P \land Q$ is true, you must (recursively) show that P is true, and also that Q is true.

Disjunction: $\lor$

Following the same reasoning,

- If we *have* an Or, that does not imply that we know *which* half of the Or is true; so we have to show that we can prove the conclusion using either half. For example, if we have (as an assumption), that x is either even or odd, then we have to show that the conclusion is true when x is even, and also that it is true when x is odd. This leads to the $\lor$-Left rule:

  $$\text{$\lor$-Left}\,\frac{\Gamma, A \vdash Q \qquad \Gamma, B \vdash Q}{\Gamma, A \lor B \vdash Q}$$

- If we *want* an Or, then we can pick whichever half to prove is more convenient. So there are actually two $\vee$-Right rules:

$$\vee\text{-Right}_1\,\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad\qquad \vee\text{-Right}_2\,\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

Implication: $\to$

For the Right rule for $\to$, we want to prove that $A \to B$ is true. To do this, we *assume* that A is true, and then show that a proof of B follows:

$$\to\text{-Right}\,\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

The $\vdash$ of a sequent is a kind of "syntactical implication", in the same way that the commas to the left of the $\vdash$ are a kind of "syntactical conjunction". So just as $\wedge$ on the left just expands into its elements, so $\to$ just splits its elements between the left and right.

If we expand out the definition of $\neg$, we have a $\neg$-Right "rule":

$$\neg\text{-Right}\,\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}$$

I.e., to prove $\neg A$, we assume A and show that a contradiction results. (This is not a real rule; it is a result of the above rules, rather than an additional rule in our system.)

The Left rule for $\to$ shows how to *apply* an implication: if we have $A \to B$ and a proof of A then we should also be able to conclude B (and then use B to conclude whatever we actually want):

$$\to\text{-Left}\,\frac{\Gamma \vdash A \qquad \Gamma, B \vdash Q}{\Gamma, A \to B \vdash Q}$$

I.e., to use $A \to B$ on the left, we must show that we can get an A from the context, and then show that adding a B (the "output" of the $A \to B$) to the context allows us to get a Q.[2]

[2] For simplicity, I've shown the contexts as if the $A \to B$ is removed from them; but remember that we can freely "copy" elements of the context, so if you still need $A \to B$ as part of proving $\Gamma \vdash A$, you can simply copy it before that point.

True and False: ⊤ and ⊥

For ⊤ (true) we only have a right rule: true is true everywhere, in any context:

$$\top\text{-Right}\ \dfrac{}{\Gamma \vdash \top}$$

For ⊥ (false) we only have a left rule: if we assume false, then *any* proposition can be proved (*ex falso quodlibet*, from false, anything follows):

$$\bot\text{-Left}\ \dfrac{}{\Gamma, \bot \vdash Q}$$

Derivations: forms of proofs

Given the above rules, we can construct some simple proofs, in the form of *derivations*. For example, we can prove that $A \wedge B \to B \wedge A$:

$$\to\text{-Right}\ \dfrac{\wedge\text{-Left}\ \dfrac{\wedge\text{-Right}\ \dfrac{\text{Assume}\ \dfrac{}{A, B \vdash A} \qquad \text{Assume}\ \dfrac{}{A, B \vdash B}}{A, B \vdash B \wedge A}}{A \wedge B \vdash B \wedge A}}{\vdash A \wedge B \to B \wedge A}$$

We work from the bottom up, starting with the theorem we wish to prove. The top-level operator of the consequent usually determines what rule to apply next.

Exercises

For all of the following, start with an empty context. (Generally speaking, your first action will be to apply the →-Right rule, giving you something in the context.)

Exercise 1:

Prove (construct a derivation showing) that

$$A \wedge B \to A \vee B$$

Exercise 2:

Prove

$$A \vee B \to B \vee A$$

Exercise 3:

Prove
$$A \wedge \top \leftrightarrow A$$
(I.e., $\top$ is the identity for $\wedge$. Remember that $P \leftrightarrow Q$ is a shorthand for $(P \rightarrow Q) \wedge (Q \rightarrow P)$.)

Exercise 4:

Prove
$$A \vee \bot \leftrightarrow A$$
(I.e., $\bot$ is the identity for $\vee$.)

Exercise 5:

Prove
$$A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$$

Exercise 6:

Prove
$$A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$$

The next few exercises require the use of the $\rightarrow$-Left rule, which is not officially part of our system.

Exercise 7:

Prove *modus ponens*:
$$(A \wedge (A \rightarrow B)) \rightarrow B$$

Exercise 8:

Prove the Law of Noncontradiction:
$$A \wedge \neg A \rightarrow \bot$$
Remember that $\neg A$ is shorthand for $A \rightarrow \bot$.

Exercise 9:

A common definition of $A \rightarrow B$ in truth-tables is

$$A \rightarrow B \quad \equiv \quad \neg A \vee B$$

Is it possible to prove this equivalence (if-and-only-if) in our system?

Exercise 10:

Similarly, in classical logic

$$\neg\neg A \rightarrow A$$

Is this provable in our system?

What about

$$\neg\neg\neg A \rightarrow \neg A$$

Exercise 11:

Proof-by-contradiction requires the use of the Law of the Excluded Middle, which states that *every proposition is either true or false*:

$$A \vee \neg A \quad \rightarrow \quad \top$$

Why is this *not* provable in our system? (In fact, the absence of LEM from our logic is *the* major difference between intuitionistic logic and classical logic.)

3 Proof Objects: Values of Type `proof`

We want to encode the structure of a proof in a value, an object of type `proof`. We need to augment our rules to describe not just when something is true, but what *value* represents the fact that something is true. We'll switch to using propositions of the form

$$p : P$$

which should be read as "p (a proof object) is a proof of the proposition P". The form that p takes will depend on the form of P. In a Right-rule, $p : P$ describes how to *construct* a proof object for P; i.e., Right-rules correspond to *constructors*. In a Left-rule for $p : P$, we describe how to *use* a proof object for P; i.e., what kind of information we can *extract* from it. Obviously, these two should align with each other; we should not be able to extract more information from a proof object than we put in during its creation, and when we construct a proof object, we should not add more information to it than it is possible to later extract.

In this section, I'll appeal to an intuition which is quite possibly the central point of these notes:

When we wish to prove $\Gamma \vdash Q$, think of the proof as a *program*, whose arguments/ inputs are given by the elements of Γ, and whose return type/output is given by Q.

When looking at a given logical construct on the Left, we think of it as an argument, something we already have and are looking to use. When looking at a given logical construct on the right, we think of it as the output of a process. Hence, a proof is an *algorithm* for transforming the assumptions in Γ into the output Q.

To drive this point home, side-by-side with the modified inference rules, we'll also write (in C++-esque pseudocode) the definition of a function

```
fun prove(theorem t, proof p, context c) -> bool ;
```

which takes a theorem (proposition), a proof, and a context as parameters, and returns True if the given proof is a valid proof of the theorem, assuming the elements of the context.

Conjunction: $\wedge$

It seems reasonable that a proof a $A \wedge B$ should include *both* a proof object for A and a proof object for B. We represent a proof object for $A \wedge B$ as a pair of proof objects, (a, b) where $a : A$ and $b : B$. Thus, we have

$$\wedge\text{-Right}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \wedge B}$$

```
fun prove(A ∧ B, (a,b), ctx) -> bool
{
    return prove(A, a, ctx) and
           prove(B, b, ctx) ;
}
```

Similarly, if we have (assume) $A \wedge B$ then we can freely extract out of it the proof of A and/or the proof of B:

$$\wedge\text{-Left}\ \frac{\Gamma, a : A, b : B \vdash q : Q}{\Gamma, (a, b) : A \wedge B \vdash q : Q}$$

```
fun prove(Q, q, [(a,b) : A ∧ B, ctx]) -> bool
{
    return prove(Q, q, [a:A, b:B, ctx]) ;
}
```

A proof object for a conjunction can be thought of as a `struct` with two members: the first a proof of A and the second a proof of B. To return/output such a `struct`, we have to include values (proofs) for both members (in logic there is no such thing as "uninitialized"!). If we receive this `struct` as an argument, then we can extract and use either, both, or neither of its members.

Disjunction: $\vee$

For disjunction, our intuition of proof-as-program comes into play. Imagine a type in a programming language `either<A,B>` which can contain either a value of type A or a value of type B (e.g., `either<int,string>` can be either an `int` or a `string`).[3] Presumably there is some operator `is` which we can use to determine *which* of the two types is currently present. This implies that internally, `either` must store not just the value of A or B but also a *tag* which records which type is currently being stored.

For us, proof objects for $P \vee Q$ will take the the form $(1, p)$ or $(2, q)$. The first element is the tag, recording which of the two disjuncts is being proved, while the second element is the proof object.

The Right rules for $\vee$ correspond to a function which *returns* a `either<A,B>`; the function has a choice: it can either construct the `either` for A or for B. So we have two Right rules:

$$\vee\text{-Right}_1\ \frac{\Gamma \vdash p : P}{\Gamma \vdash (1, p) : P \vee Q}$$

```
fun prove(P ∨ Q, (1,p), ctx) -> bool
{
    return prove(P, p, ctx) ;
}
```

$$\vee\text{-Right}_2\ \frac{\Gamma \vdash q : Q}{\Gamma \vdash (2, q) : P \vee Q}$$

```
fun prove(P ∨ Q, (2,q), ctx) -> bool
{
    return prove(Q, q, ctx) ;
}
```

[3]In C++ `either` is known as `variant<...>` and it can take any number of types, not just two.

On the left hand side, imagine a function which has an *argument* of type `either<A,B>`. Now, the body of the function probably looks something like this:

```
fun f(either<A,B> e) -> G
{
    if(e is A) {
        // Extract a:A from e
        // .. Something to turn an a:A into a g₁:G
    }
    else { // e is B
        // Extract b:B from e
        // .. Something to turn a b:B into a g₂:G
    }
}
```

Thus, to use a $\vee$ on the Left, we must supply a kind of if/else structure, showing how we can take *either* alternative and make them both work. We call this structure `case`:

$$\vee_L \frac{\Gamma, p : P \vdash g_1 : G \qquad \Gamma, q : Q \vdash g_2 : G}{\Gamma, e : P \vee Q \vdash \mathsf{case}(e) \begin{cases} g_1 & \text{if } e = (1, p) \\ g_2 & \text{if } e = (2, q) \end{cases} : G}$$

```
fun prove(G, case(g₁,g₂), [e : P ∨ Q, ctx]) -> bool
{
    if(e == (1,p))
        return prove(G, g1, [p:P, ctx]) ;
    else // e == (2,q)
        return prove(G, g2, [q:Q, ctx]) ;
}
```

This rule says, if we have a $e : P \vee Q$ on the left, then to use it we have to show that

- If $e = (1, p)$ we can use $p : P$ to produce a proof $g_1 : G$ of our consequent.
- If $e = (2, q)$ w ecan use $q : Q$ to produce a proof $g_2 : G$ of our consequent. Note that g_1 and g_2 may be (most likely are) different, because they were constructed from different assumptions.

The `case` construct just bundles up all the relevant elements.

It's worth considering what would happen if proof objects for $\lor$ were not "labeled" with which "side" of the $\lor$ they came from. In dynamically-typed programming languages such as Python and Javascript, a variable's type can change at runtime: a variable might be a string at one moment, an array the next, and a number after that. However, there is always a way to query what the *current* type of the object is. In Javascript this is `typeof`, in Python it's `isinstance`. This tells us that, under the hood, these programming language are storing not just the *value* but also a "tag". I.e., the value of a variable containing (say) a string is not *just* the bits necessary to represent the string, but also an additional value indicating "this variable contains a string". This is the situation we are in with our "tagged" $\lor$ proof objects.

If we omit the tags, and allow proofs of $P \lor Q$ to just be a naked $p : P$ or $q : Q$ then we are in a situation more like assembly language, where a given sequence of bits can be a string, or an integer (signed? unsigned? biased?), or a floating-point value, or part of an array, or an address, and *we have no idea which one it is*. Given a proof object $e : P \lor Q$ we cannot extract *any* information from it.

This is also why we only allow one consequent on the right side of the $\vdash$; to prove $\Gamma \vdash P, Q$ we only have to show that $\Gamma \vdash P$ or $\Gamma \vdash Q$ and we are not required to record which way we did it. Non-intuitionistic sequents correspond to "untagged" $\lor$ proof-objects; they discard information about how a $\lor$ was proved.

Implication: $\to$

Following our intuition, a proof object for $P \to Q$ is a *function*, which takes as a parameter $p : P$ and uses it to construct and return a proof object $q : Q$, which we write as $\text{fun}(p) = q$; i.e., q represents the "body" of the function, which may use the parameter p.

$$\to_R \frac{\Gamma, p : P \vdash q : Q}{\Gamma \vdash (\text{fun}(p) = q) : P \to Q}$$

```
fun prove(P → Q, (fun(p) = q) , ctx) -> bool
{
    return prove(Q, q, [p:P, ctx]) ;
}
```

q corresponds to the "body" of the function, and is allowed to refer to the "parameter" p.

If the Right-rule for $\to$ tells us how to *construct* a function, then the Left-rule tells us how to *call* a function.

$$\to_L \frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash (b = [a/x]e) : B \\ \Gamma, b : B \vdash q : Q\end{array}}{\Gamma, (\text{fun}(x) = e) : A \to B \vdash q : Q}$$

This rule says that, to use a function $f : A \rightarrow B$, we need to show that

- We have (can derive from the context Γ) an $a : A$ to serve as its argument
- "Calling" the function with a as its parameter causes it to "return" some proof object for B. This works by *substituting* a for x within the "body" of the function e.
- With $b : B$ in hand (added to the context), we can derive the actual consequent $q : Q$.

For example, what does the proof object look like for the proposition $A \wedge B \rightarrow B \wedge A$? The proof of this will be a function which takes an object $(a, b) : A \wedge B$ and returns an object $(b, a) : B \wedge A$. The definition of this function is

```
fun((a,b)) = (b,a)
```

True and False: ⊤ and ⊥

⊤ only has a Right-rule: True is True everywhere, in any context, so we can create a proof object for True given *no* information. The proof object for ⊤ is written () and is called *unit*. It represents a completely uninteresting object: no information went into its creation, so no information can ever be extracted from it.

$$\top\text{-Right}\,\frac{}{\Gamma \vdash () : \top}$$

```
fun prove(⊤, (), ctx) -> bool
{
    return true ;
}
```

If $() : \top$ represents a completely uninteresting object, containing no information, then the proof-object for ⊥ is the opposite: a magical object which contains *all knowledge*. This corresponds to the classical principle *ex falso quodlibet*, from false, anything follows. If we can get a ⊥ on the left, then *any* proposition can be proved from it!

In computational terms, this is somewhat confusing: we have a function which takes as an argument, a value which cannot exist! But then it can magically turn that value into *any* proof object necessary.

$$\bot\text{-Left}\,\frac{}{\Gamma, e : \bot \vdash \mathtt{err}(e) : Q}$$

```
fun prove(Q, err(e₁), [e₂ : ⊥, ctx]) -> bool
{
    return e₁ == e₂ ;
}
```

`err`(e) is a “magical” proof object: it can serve as the proof of *any* proposition, if only you can manage to construct it!

Figure 2 gives a summary of these rules.

Examples

What is the proof object for $A \wedge B \rightarrow A \vee B$? There are two distinct proofs, corresponding to the deriviations:

$$\begin{prooftree}
\AxiomC{}
\LeftLabel{Assume}
\UnaryInfC{$A, B \vdash A$}
\LeftLabel{$\vee$-Right$_1$}
\UnaryInfC{$A, B \vdash A \vee B$}
\LeftLabel{$\wedge$-Left}
\UnaryInfC{$A \wedge B \vdash A \vee B$}
\LeftLabel{$\rightarrow$-Right}
\UnaryInfC{$(\texttt{fun}((a, b)) = (1, a)) : A \wedge B \rightarrow A \vee B$}
\end{prooftree}$$

$$\begin{prooftree}
\AxiomC{}
\LeftLabel{Assume}
\UnaryInfC{$a : A, b : B \vdash b : B$}
\LeftLabel{$\vee$-Right$_2$}
\UnaryInfC{$a : A, b : B \vdash (2, b) : A \vee B$}
\LeftLabel{$\wedge$-Left}
\UnaryInfC{$(a, b) : A \wedge B \vdash (2, b) : A \vee B$}
\LeftLabel{$\rightarrow$-Right}
\UnaryInfC{$(\texttt{fun}((a, b)) = (2, b)) : A \wedge B \rightarrow A \vee B$}
\end{prooftree}$$

Exercises

Exercise 12:

For each (provable) exercise in the first section, construct its corresponding proof object.

4 From Propositions to Predicates

Currently, the building-blocks of our logic are the bare propositions $A, B, C, \ldots$. These represent simple “facts”, true in and of themselves. In order to do anything interesting, we must extend our logic to *predicates* of the form $P(x)$, stating that some property P is true *of the object* x. That is, we now deal in theorems about the *properties of values*. Extending our logic to predicates also requires us add universal ($\forall$) and existential ($\exists$) quantification rules.

A n-ary *predicate* is a proposition of the form

$$P(t_1, t_2, \ldots, t_n)$$

where the t_i are the *arguments* of the predicate. A predicate is a relation which is true for some assignments of values to its arguments. For example, we can define a predicate $\mathsf{nat}(x)$ which is true if-and-only-if x is a natural number (i.e., an integer ≥ 0):

$$\text{Nat-0}\,\frac{}{\Gamma \vdash \mathsf{nat}(0)} \qquad \text{Nat-S}\,\frac{\Gamma \vdash \mathsf{nat}(x)}{\Gamma \vdash \mathsf{nat}(s(x))}$$

These rules state that 0 is a natural number, and $s(x)$ is a natural number if-and-only-if x is also a natural number (i.e., this is the classic inductive definition of $\mathbb{N}$).

Many things that we'd normally think of as *operations* will be expressed instead as predicates. For example, to define addition of natural numbers, we will define a predicate corresponding to the equation $a + b = c$, written as $\mathsf{add}(a, b, c)$:

$$\text{Add-0}\,\frac{}{\Gamma \vdash \mathsf{add}(0, x, x)} \qquad \text{(Base case: } 0 + x = x\text{)}$$

$$\text{Add-S}\,\frac{\Gamma \vdash \mathsf{add}(x, y, z)}{\Gamma \vdash \mathsf{add}(s(x), y, s(z))} \qquad \text{(Recursive case: } (1 + x) + y = (1 + z) \text{ if } x + y = z\text{)}$$

Using this definition, here's a derivation showing that $2 + 3 = 5$:

$$\text{Add-S}\,\dfrac{\text{Add-S}\,\dfrac{\text{Add-Z}\,\dfrac{}{\mathsf{add}(0, s(s(s(0))), s(s(s(0))))}}{\mathsf{add}(s(0), s(s(s(0))), s(s(s(s(0)))))}}{\mathsf{add}(s(s(0)), s(s(s(0))), s(s(s(s(s(0))))))}$$

In the future, we'll write natural numbers in their normal decimal form, with the understanding that $1 \equiv s(0), 2 \equiv s(s(0)), \ldots$ Using this more readable form, here's the above derivation:

$$\text{Add-S}\,\dfrac{\text{Add-S}\,\dfrac{\text{Add-Z}\,\dfrac{}{\mathsf{add}(0, 3, 3)}}{\mathsf{add}(1, 2, 4)}}{\mathsf{add}(2, 3, 5)}$$

I.e., $2 + 3 = 5$ *because* $1 + 3 = 4$ *because* $0 + 3 = 3$ (base case).

I've gone to the trouble of defining natural numbers not just for fun, but also because we need something for predicates to be *about*.

Universal and Existential Quantification

coming soon...

5 Appendix A: Summary of inference rules

$$\text{Assume}\,\frac{}{P_1, \ldots, P_n \vdash P_i}$$

$$\top_R\,\frac{}{\Gamma \vdash \top} \qquad \bot_L\,\frac{}{\Gamma, \bot \vdash Q}$$

$$\wedge_R\,\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \qquad \wedge_L\,\frac{\Gamma, P, Q \vdash G}{\Gamma, P \wedge Q \vdash G}$$

$$\vee_{R_1}\,\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee_{R_2}\,\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

$$\vee_L\,\frac{\Gamma, P \vdash G \qquad \Gamma, Q \vdash G}{\Gamma, P \vee Q \vdash G}$$

$$\to_R\,\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q} \qquad \to_L\,\frac{\Gamma, A \to B \vdash A \qquad \Gamma, A \to B, B \vdash Q}{\Gamma, A \to B \vdash Q}$$

Figure 1: Rules without proof objects

$$\text{Assume}\,\frac{}{p_1 : P_1, \ldots, p_n : P_n \vdash p_i : P_i}$$

$$\top_R\,\frac{}{\Gamma \vdash () : \top} \qquad \bot_L\,\frac{}{\Gamma, e : \bot \vdash \mathsf{err}(e) : P}$$

$$\land_R\,\frac{\Gamma \vdash p : P \qquad \Gamma \vdash q : Q}{\Gamma \vdash (p, q) : P \land Q} \qquad \land_L\,\frac{\Gamma, p : P, q : Q \vdash G}{\Gamma, (p, q) : P \land Q \vdash G}$$

$$\lor_{R_1}\,\frac{\Gamma \vdash p : P}{\Gamma \vdash (1, p) : P \lor Q} \qquad \lor_{R_2}\,\frac{\Gamma \vdash q : Q}{\Gamma \vdash (2, q) : P \lor Q}$$

$$\lor_L\,\frac{\Gamma, p : P \vdash g_1 : G \qquad \Gamma, q : Q \vdash g_2 : G}{\Gamma, e : P \lor Q \vdash \mathsf{case}(e) = \begin{cases} g_1 & \text{if } e = (1, p) \\ g_2 & \text{if } e = (2, q) \end{cases} : G}$$

$$\to_R\,\frac{\Gamma, p : P \vdash e : Q}{\Gamma, \vdash (\mathsf{fun}(p) = e) : P \to Q}$$

$$\to_L\,\frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma, [a/x]e : B \vdash q : Q\end{array}}{\Gamma, (\mathsf{fun}(x) = e) : A \to B \vdash q : Q}$$

Figure 2: Rules with proof objects

6 Appendix B: About this Document

This document was typeset in LaTeX, using the `memoir` document class.

Body text is set in TeX Gyre Pagella, an open-source version of Palatino Linotype.

Sans text is set in Andika New Basic.

Math is set in GFS Neohellenic.

Computer type is set in `Iosevka`.

The following packages were used (this list is not exhaustive):

- `booktabs`, for nicer-looking tables
- `exercises`, for (automatically numbered) exercises
- `multicol`, for multi-column layouts
- `semantic`, for inference rules.
- `TikZ`, for diagrams