Inductive Definitions

CPSC 509: Programming Language Principles

Ronald Garcia*

8 January 2014
(**Time Stamp**: 13:15, Thursday 3rd February, 2022)

In the last set of notes, you got a taste of some of the kinds of models that we will build (of language syntax and semantics) in the context of a very very simple language—featuring a finite number of programs *and* results. In these notes we start to develop the machinery for building more complex languages, starting with a language that at least features an *infinite* number of programs (still finite results though!). The tools we introduce here, called *inductive definitions* will be critical throughout the course, and we'll find later that they connect directly to what you may have learned about *proof by induction* in prior courses.

1 Inductive Definitions

Now we are going to look at a somewhat more interesting language. The language will have an infinite number of programs. This means that we can't simply list all of the programs: we need a new strategy.

The way that we define our set of programs follows a pattern similar to what happens in a real language implementation. Consider how a parser for the C programming language works. It takes as input any finite *strings* (i.e., sequences) of *bytes* (of which there are $2^8 = 256$), and filters out the strings that count as legal C programs.[1]

$$\text{CPGMS} = \{\, s \in \text{BYTE}^* \mid s \text{ is a valid C program} \,\}$$

Here BYTE is the set of bytes, and in general, we say that S is some set, then S^* is the set of all finite sequences of elements of S (including the empty sequence, which for visibility's sake we denote by ε). So essentially, the set of C programs is defined by filtering all the valid C programs out of the set of all byte strings.

The key idea here is that we start with some basic, easily defined pre-existing set and filter it down to the subset that we want. Most mathematical definitions are based around this general idea.

We do not want to concern ourselves with as low level a representation as strings of bytes, so we start more abstractly. First, we assume some set of *atomic* elements.

$$a \in \text{ATOM}$$

Our base assumption is that there are an infinite number of ATOMs in the set (not just 256), so we can always find another one if we need one.[2] Our only requirement of ATOMs is that we can *tell them apart*: we don't care what they really look like, just given two of them we can tell if they are the same or not. We can state this formally as a (seemingly obvious) property:

Proposition 1. $\forall a_1, a_2 \in \text{ATOM}. a_1 = a_2 \vee a_1 \neq a_2$.

[1] You wouldn't want your C compiler to segfault if you gave it a file of random noise, right? It should reject the program and exit gracefully.

[2] To make this concrete, it's as though we had a computer that could work with arbitrary natural numbers instead of bytes.

For our purposes, this should be interpreted as saying "given atoms a_1 and a_2, we can determine whether they are equal or not." That reading may be a bit different than you are used to: don't worry about that for now. These ATOMs will be used to represent the primitive constructs of our languages. If you've ever studied parsing, this is analogous to your tokens. For purposes of reasoning, we'll give each atom an abstract name, written in blue, like car, or avocado-toast, and whenever we use two different abstract names, we are referring to two different atoms, e.g. car $\neq$ avocado-toast by convention. Note that this is *very* different than referring to atoms using *metavariables* like a_1 or a_2. We cannot up-front assume that $a_1 \neq a_2$...only if we have that fact as an assumption (or can prove it in some context).

Then, building on top of the ATOMs, we assume a set of all possible trees of ATOMs.

$$r \in \textsc{Tree}[\textsc{Atom}]$$

In set theory, we can create pairs of elements of sets, subsets of sets, and so forth. Using the basic tools of set theory, we can design a representation for trees, just like we can build tree data structures in the programming language of your choice. We're not going to actually bother building up a particular set-based representation of trees, but just keep in mind that you could come up with one, given some time and patience. In this case TREE[ATOM] stands for the set of all trees that have atoms at their nodes. In general, we write TREE$[X]$ for "trees of elements of X." For now, since we are only concerned with trees of atoms, we will just write TREE as an abbreviation.

These TREEs have ATOMs for nodes. For convenience, we can write them in parenthesized notation, e.g., $a_1(a_2, a_3)$ is the TREE with parent node a_1 and two children a_2 and a_3.[3] This corresponds to the TREE:

Now, armed with a set of TREEs, we will define the *abstract syntax* of our language. The general definition strategy is to isolate a set of *terms*, which will be some subset of the set of all TREEs:

$$t \in \textsc{Term}, \text{ where } \textsc{Term} = \{\, r \in \textsc{Tree} \mid P(r) \,\} \tag{1}$$

and P is some property of TREEs that chooses the ones that we want to consider to be terms. It's our job, then, to specify that property.

First we give convenient names to some ATOMs that we'll use in our language definition:

$$\text{Assume } \{\, \mathsf{true}, \mathsf{false}, \mathsf{if} \,\} \subseteq \textsc{Atom}.$$

Then we carve out our set of TERMs using what are called *inductive rules*.

$\boxed{\textsc{Term} \subseteq \textsc{Tree}}$

$$\frac{}{\mathsf{true}}\ (\text{rtrue}) \qquad \frac{}{\mathsf{false}}\ (\text{rfalse}) \qquad \frac{r_1 \quad r_2 \quad r_3}{\mathsf{if}(r_1, r_2, r_3)}\ (\text{rif})$$

What you see above are three *rules* which together will help us define the set of TERMs. Each rule has zero or more *premises* above the horizontal bar, and one *conclusion* below it. Next to each rule is a *name* in parentheses. In essence, each of these rules can be informally read as saying "if all of the premises are members of the set, then the conclusion is as well."

Technically, a rule stands for the set of all of its instances. For example:

$$(\text{rif}) = \left\{ \frac{r_1 \quad r_2 \quad r_3}{\mathsf{if}(r_1, r_2, r_3)} \in \textsc{Rule}[\textsc{Tree}] \;\middle|\; r_1 \in \textsc{Tree}, r_2 \in \textsc{Tree}, r_2 \in \textsc{Tree} \right\}.$$

Really, pushing down even further, each rule instance is just a pair $\langle X, x \rangle \in \mathcal{P}(\textsc{Tree}) \times \textsc{Tree}$ of a set of trees and a single tree: the premises (in no particular order) and the conclusion. So *really* the rule above is interpreted as follows:

$$(\text{rif}) = \{\, \langle \{\, r_1, r_2, r_3 \,\}, \mathsf{if}(r_1, r_2, r_3) \rangle \in \mathcal{P}(\textsc{Tree}) \times \textsc{Tree} \mid r_1 \in \textsc{Tree}, r_2 \in \textsc{Tree}, r_2 \in \textsc{Tree} \,\}.$$

[3] A more precise notation for this tree would be $a_1(a_2(), a_3())$ but we will elide empty parentheses for childless nodes.

Each *judgment* (i.e. each premise and each conclusion) in a rule represents a TREE, and each rule stands for a number of *instances* of rules. An instance of a rule is the result of replacing the black italic *metavariables* (e.g., r_1) with a concrete TREE. Here is an example of an unsurprising instance of the (rif) rule:

$$\frac{\mathsf{true} \quad \mathsf{false} \quad \mathsf{true}}{\mathsf{if}(\mathsf{true}, \mathsf{false}, \mathsf{true})}\ (\text{rif})$$

However, the rules also imply "nonsensical" rule instances, since *any* concrete TREE can be placed where there is a metavariable r. For example, if we assume that there exist ATOMs named under, and while, then the following is a perfectly fine instance of the (rif) rule:

$$\frac{\mathsf{under} \quad \mathsf{while} \quad \mathsf{while}}{\mathsf{if}(\mathsf{under}, \mathsf{while}, \mathsf{while})}\ (\text{rif})$$

Notice that even the nonsensical rule instance we wrote above makes sense under this intuitive interpretation: if under and while *were* TERMs, then the TREE in the conclusion would also be a term; but they are not, so it is not.

We now introduce two forms of *syntactic sugar* that are common in books and papers that use this technique to define sets. Both of these notational devices will make our representation look a little prettier, but otherwise add no real content. To make the syntax of our language look a bit more like a real programming language, we will write the TERM

$$\mathsf{if}(t_1, t_2, t_3)$$

using the notation

$$\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3.$$

In general we will often introduce expressions where the ATOM "name" is split between the different arguments. This is often called *mixed-fix* notation (as opposed to the *prefix* notation we were using above).

The second syntactic nicety we will add is that we'll write the rules so that they refer to the name of the set that we are defining. For example, we'll rewrite the rules as follows:

$$\frac{}{\mathsf{true} \in \text{TERM}}\ (\text{rtrue}) \qquad \frac{}{\mathsf{false} \in \text{TERM}}\ (\text{rfalse}) \qquad \frac{r_1 \in \text{TERM} \quad r_2 \in \text{TERM} \quad r_3 \in \text{TERM}}{\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \text{TERM}}\ (\text{rif})$$

Going forward, it helps to remember that the "$\in$ TERM" part is really just a decoration for our human purposes. Think of it like a comment in a programming language (i.e., in C you would write `/* in Term */`).

With the syntactic sugar out of the way, we still have to discuss how these rules define our set of TERMs. For some insight, consider the (rtrue) rule. According to our interpretation of rules, this rule says that with no premises whatsoever, true is a TERM. The (rfalse) rule is similar. Generalizing this observation, we can show in multiple steps that some TREE is a TERM under no premises by building up a *derivation tree* of rule instances that as a whole has no premises, meaning that every rule instance at the top of the derivation tree is closed with a horizontal bar that has nothing above it. Here is an example of a derivation tree $\mathcal{D}$, built from rule instances, that shows that if true then false else true $\in$ TERM:

$$\mathcal{D} = \dfrac{\dfrac{}{\mathsf{true} \in \text{TERM}}\ (\text{rtrue}) \quad \dfrac{}{\mathsf{false} \in \text{TERM}}\ (\text{rfalse}) \quad \dfrac{}{\mathsf{true} \in \text{TERM}}\ (\text{rtrue})}{\mathsf{if\ true\ then\ false\ else\ true} \in \text{TERM}}\ (\text{rif})$$

What we did was instantiate the (rif) rule such that $r_1 = \mathsf{true}$, $r_2 = \mathsf{false}$, and $r_3 = \mathsf{true}$, giving:

$$\frac{\mathsf{true} \in \text{TERM} \quad \mathsf{false} \in \text{TERM} \quad \mathsf{true} \in \text{TERM}}{\mathsf{if\ true\ then\ false\ else\ true} \in \text{TERM}}\ (\text{rif}).$$

Since a derivation can't have open premises, we had to consider each of the three premises in turn, but we were able to find closed derivations for each of them using the (rtrue) and (rfalse) rules. As above, we'll use uppercase calligraphic letters like $\mathcal{D}, \mathcal{E}, \mathcal{F}$ as metavariables that stand for derivation trees, and we write both

$$\begin{matrix}\mathcal{D} \\ t \in \text{TERM}\end{matrix}$$

and

$$\mathcal{D} :: t \in \textsc{Term}$$

to mean that $\mathcal{D}$ is a derivation of $t \in \textsc{Term}$.

Technically, we collect all of the rule instances into a *rule set* $\mathcal{R} = (\text{rtrue}) \cup (\text{rfalse}) \cup (\text{rif})$ and use it to induce the set of derivations $\mathcal{D} \in \textsc{Deriv}[\mathcal{R}] \subseteq \textsc{Tree}$ that can be constructed using the rules $\mathcal{R}$.

In this case we can refer the set of derivations of $t \in \textsc{Term}$ by the name $\textsc{Deriv}[\mathcal{R}]$, to mean the set of all derivations that can be built using the rules listed in square brackets. That set name is quite a mouthful, so for brevity, just like with TREEs, we'll just say DERIV and infer from context which rule set $\mathcal{R}$ we have in mind.

These derivations are the piece we've been building toward to define the set of TERMs. Notice that no matter what TREEs r_i we instantiate the (rif) rule with, the conclusion will be a TREE. Technically the (rtrue) and (rfalse) rules each have only one instantiation (because they have no TREE metavariables), each of which is a TREE. We can thus define the set of TERMs to be the subset of TREEs that have derivations according to our rules, or in formal terms:

$$\textsc{Term} = \{\, r \in \textsc{Tree} \mid \exists \mathcal{D} \in \textsc{Deriv}[\mathcal{R}].\mathcal{D} :: r \in \textsc{Term} \,\}.$$

i.e., $P(r) \equiv \exists \mathcal{D} \in \textsc{Deriv}.\mathcal{D} :: r \in \textsc{Term}$ in Equation (1) above.

In the particular case of defining the abstract syntax of a language, we have at our disposal the well-known Backus-Naur Form (BNF), to succinctly capture inductive definitions of syntax.

$$\begin{array}{l} t \in \textsc{Term}, \\ t \quad ::= \quad \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \end{array}$$

The grammar above is shorthand for the inductive rules that we gave above. In the future, we'll use BNF to specify our syntax, but we'll end up using inductive rules to define other sets and relations. This kind of set definition is in general called an *inductive definition*.

2 Example: Propositional Logic

To help affirm the ideas introduced above, we will introduce a couple of new definitions. The first is of an abstract syntax, while the second is of a binary relation (which, recall, is a set of pairs).

What we're going to do is introduce a definition of *a* logic. I emphasize the idea of *a* logic, because there are many different logical systems in the world. In particular, we will define what is called *constructive propositional logic* or *intuitionistic propositional logic*. We may talk about what makes it constructive later on.

This example serves two purposes for us. The first is to provide another example of an inductive definition. The second purpose is to introduce a *formal* model of a small piece of logic, including propositions and proofs, that corresponds to the kind of more *informal* logical arguments that we will be using to state and prove properties of programming languages. Our informal arguments will be a bit looser, but ultimately we could fully formalize them with a substantial amount of effort. In fact, recent advances in *mechanized proof assistants* allows you to do exactly that: write down propositions in a computer language, write proofs of them, and have the computer verify that the proofs are correct. In some cases we can even automate the generation of those proofs. For this class, though, we will focus on the more informal approach, but write our proofs so that there is enough detail that we could in principle fully formalize them, and maybe even mechanize them.

2.1 Syntax

First, we'll start with the syntax of constructive propositional logic. Here's the BNF first:

$$\begin{array}{l} A \in \textsc{AtomicProp}, \quad p \in \textsc{Prop}, \\ p \quad ::= \quad A \mid \top \mid \bot \mid p \wedge p \mid p \vee p \mid p \supset p \end{array}$$

The above corresponds exactly to the following:

First, assume some set $A \in \text{ATOMICPROP}$.

Given this, we define the set $p \in \text{PROP}$ using the following rules:

$\boxed{\text{PROP} \subseteq \text{TREE}}$

$$\frac{}{A \in \text{PROP}}\ (\text{atm}) \qquad \frac{}{\top \in \text{PROP}}\ (\text{top}) \qquad \frac{}{\bot \in \text{PROP}}\ (\text{bot}) \qquad \frac{r_1 \in \text{PROP} \quad r_2 \in \text{PROP}}{r_1 \wedge r_2 \in \text{PROP}}\ (\text{and})$$

$$\frac{r_1 \in \text{PROP} \quad r_2 \in \text{PROP}}{r_1 \vee r_2 \in \text{PROP}}\ (\text{or}) \qquad \frac{r_1 \in \text{PROP} \quad r_2 \in \text{PROP}}{r_1 \Rightarrow r_2 \in \text{PROP}}\ (\text{imp})$$

Notice that the BNF uses the metavariable p to define the set PROP, while the inductive rules use the metavariable r to explicitly define the set of derivation trees as instances of the above rules. The BNF version is just a convention, but the "real" thing to do is define it in terms of some pre-existing set: in our case TREEs.

Now let me briefly explain the notation. Our language has arbitrary atomic propositional constants, $\top$ and $\bot$, which correspond to logical truth and falsity, $\wedge$ and $\vee$, which correspond to logical conjunction and disjunction (i.e., "and" and "or"), and $\supset$ which is logical implication, (i.e., "if-then"). Notice that the language does not define negation (i.e., "not"). We represent negation using implication and falsehood: in particular "A is not true" is represented by "A implies falsehood," in symbols $A \Rightarrow \bot$. However, for convenience we provide some syntactic sugar: $\neg p \equiv p \Rightarrow \bot$.

Here is an example of a proposition. Let's let the following atomic propositions have the given meanings:

$$\begin{aligned} \mathsf{C} &\equiv \text{Chocolate is delicious} \\ \mathsf{A} &\equiv \text{Apples are delicious} \\ \mathsf{G} &\equiv \text{Apples are good for you} \\ \mathsf{S} &\equiv \text{Chocolate is good for you} \end{aligned}$$

Then the proposition:

$$\mathsf{C} \wedge \mathsf{A} \wedge \mathsf{G} \supset \mathsf{S}$$

is the (obviously correct) statement "If chocolate is delicious and apples are delicious and apples are good for you, then chocolate is good for you."

2.2 Entailment and a Discussion of Side-conditions

Now that we have defined the set of propositional formulas, we can establish what counts as proofs of propositions. In actuality, what we will define is a relation called *entailment*, which establishes the following: given the knowledge of certain truths, what other truths can we deduce (i.e., what truths are entailed by other truths)?

We formalize this as a binary relation.

First let $\Gamma \in \mathcal{P}(\text{PROP})$. Then we define the binary relation $\cdot \vdash \cdot$ **true** $\subseteq \mathcal{P}(\text{PROP}) \times \text{PROP}$ using the following inductive rules:

$\boxed{(\cdot \vdash \cdot \ \mathbf{true}) \subseteq \mathcal{P}(\textsc{Prop}) \times \textsc{Prop}}$

$$\frac{}{\Gamma \vdash p \ \mathbf{true}} \ (\text{hyp}) \quad p \in \Gamma$$

$$\frac{}{\Gamma \vdash \top \ \mathbf{true}} \ (\top\text{I}) \qquad \frac{\Gamma \vdash \bot \ \mathbf{true}}{\Gamma \vdash p \ \mathbf{true}} \ (\bot\text{E})$$

$$\frac{\Gamma \vdash p_1 \ \mathbf{true} \quad \Gamma \vdash p_2 \ \mathbf{true}}{\Gamma \vdash p_1 \wedge p_2 \ \mathbf{true}} \ (\wedge\text{I}) \qquad \frac{\Gamma \vdash p_1 \wedge p_2 \ \mathbf{true}}{\Gamma \vdash p_1 \ \mathbf{true}} \ (\wedge\text{E1}) \qquad \frac{\Gamma \vdash p_1 \wedge p_2 \ \mathbf{true}}{\Gamma \vdash p_2 \ \mathbf{true}} \ (\wedge\text{E2})$$

$$\frac{\Gamma \vdash p_1 \ \mathbf{true}}{\Gamma \vdash p_1 \vee p_2 \ \mathbf{true}} \ (\vee\text{I1}) \qquad \frac{\Gamma \vdash p_2 \ \mathbf{true}}{\Gamma \vdash p_1 \vee p_2 \ \mathbf{true}} \ (\vee\text{I2})$$

$$\frac{\Gamma \vdash p_1 \vee p_2 \ \mathbf{true} \quad \Gamma \cup \{ p_1 \} \vdash p_3 \ \mathbf{true} \quad \Gamma \cup \{ p_2 \} \vdash p_3 \ \mathbf{true}}{\Gamma \vdash p_3 \ \mathbf{true}} \ (\vee\text{E})$$

$$\frac{\Gamma \cup \{ p_1 \} \vdash p_2 \ \mathbf{true}}{\Gamma \vdash p_1 \supset p_2 \ \mathbf{true}} \ (\supset\text{I}) \qquad \frac{\Gamma \vdash p_1 \supset p_2 \ \mathbf{true} \quad \Gamma \vdash p_1 \ \mathbf{true}}{\Gamma \vdash p_2 \ \mathbf{true}} \ (\supset\text{E})$$

The (hyp) rule captures the idea that if you have assumed a property is true, then you can immediately deduce that property. This rule has a slightly different form than what we have seen before: there is a constraint $p \in \Gamma$ next to the rule itself. This is called a *side-condition*. Here's how it works: the hyp rule stands for a bunch of concrete instantiations of the rule

$$\frac{}{\Gamma \vdash p \ \mathbf{true}} \ (\text{hyp})$$

However, if we leave the rule as-is, then the following instance would be legal:

$$\frac{}{\emptyset \vdash \mathsf{A} \ \mathbf{true}} \ (\text{hyp})$$

But we definitely don't want instances like that! If we allowed it, then we could prove any proposition true (under no assumptions) immediately. Instead, we use the side condition to *restrict* the instances of the rule that we will allow. We only allow instances where the proposition appears in Γ, like:

$$\frac{}{\{ \mathsf{A}, \mathsf{B} \} \vdash \mathsf{A} \ \mathbf{true}} \ (\text{hyp})$$

Desugaring the original rule into set notation gives us

$$(\text{hyp}) = \left\{ \frac{}{\Gamma \vdash p \ \mathbf{true}} \;\middle|\; p \in \Gamma \right\}$$

Notice how the side condition plays the role of a predicate in a set comprehension. This is different from a premise, which would appear to the left of the vertical bar in the set comprehension.

So side-conditions cut down the legal rule instances. As you can see, side-conditions are different from rule premises. Premises serve to generate some initial set of rule instances, by substituting relevant set elements for each metavariable. Then, the side condition acts as a filter on this large set of generated rule instances. Nonetheless, many papers and books typeset side-conditions on top of the horizontal bar as if they too were premises. For example:

$$\frac{p \in \Gamma}{\Gamma \vdash p \ \mathbf{true}} \ (\text{hyp})$$

Here, $p \in \Gamma$ is a side-condition. It is clearly not an expression that stands for a set of trees. Instead it ensures that we will only consider rule instances where the logical proposition $p \in \Gamma$ holds for the specific choice of p and Γ.

Usually you can tell from the context whether such a thing is a premise or a side-condition. Informally, you can read a side condition as just another precondition for the conclusion to be true, just like we do with

the premises; formally, however, side-conditions have a different meaning than premises. Later, we'll see that our reasoning principles for inductive definitions treat side-conditions differently than premises.

For comparison, and as another example, observe that we can replace the rule above with a different rule that represents exactly the same instances, but in this rule, the side-condition isn't as obvious:

$$\frac{}{\Gamma \cup \{\, p \,\} \vdash p \ \textbf{true}} \ \text{(hyp)}$$

Note how we're using explicit set union notation to add the proposition p to whatever set Γ that we choose to instantiate this rule with. So if we chose to instantiate that rule with $\mathsf{p_0}$ for p and $\emptyset$ for Γ, then we would get

$$\frac{}{\{\, \mathsf{p_0} \,\} \vdash \mathsf{p_0} \ \textbf{true}} \ \text{(hyp)}$$

We will often use a known function (like set-union) as part of the definition of an inductive rule. In that case it really plays the role of a side-condition, so the above rule can be read to mean exactly:

$$\frac{}{\Gamma_0 \vdash p \ \textbf{true}} \ \text{(hyp)} \quad \Gamma_0 = \Gamma \cup \{\, p \,\}$$

It's worth knowing that we can present *any* rule in a form that pushes all of the structural information about premises and conclusions into side-conditions. For instance, here is an equivalent presentation of (rif) using side-conditions:

$$\frac{r_1 \in \textsc{Term} \quad r_2 \in \textsc{Term} \quad r_3 \in \textsc{Term}}{r_4 \in \textsc{Term}} \ \text{(rif)} \quad r_4 = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3$$

This rule has *exactly* the same instances as the original. The only difference is that we've made explicit that the structure of the conclusion is in essence a side-condition. It is useful to know that multiple equivalent presentations of a rule are possible, both for designing inductive definitions to be clear, as well as understanding an inductive definition found in the wild.

Finally, let me give another example of how side-conditions and structure can affect an inductive definition:

Suppose you're looking at an inductive definition of some set $G \subseteq \mathbb{N}$, which has the following rule:

$$\frac{n_1 \quad n_2}{(n_1 \div n_2)} \ \text{(div)}$$

where $\cdot \div \cdot : \mathbb{R} \times \mathbb{R} \rightharpoonup \mathbb{R}$ is division over the real numbers.

Then the (div) rule represents the set of instances

$$\left\{ \frac{n_1 \quad n_2}{n_3} \;\middle|\; n_1, n_2, n_3 \in \mathbb{N} \wedge n_3 = n_1 \div n_2 \right\}.$$

Notice how using a mathematical function in the conclusion is essentially equivalent to having a side-condition. In fact you can always rewrite ANY rule so that each premise and each conclusion is a distinct metavariable from the set that you are refining (the set of "judgments") and all structure is relegated to side conditions. This notation would translate to an equivalent set of rule instances. Anyway, there is no instance in the set (div) with the premises $n_1 = 1$ and $n_2 = 2$ because there is no natural number n_3 that results from $1 \div 2$ (the answer is a real number that happens to also be a rational number but not a natural number). For similar but slightly different reasons, there is no instance in (div) with $n_2 = 0$, also because there is no natural number n_3 that results from $n_1 \div 0$ for any natural number n_1 (because $n_1 \div 0$ is undefined for all such cases).

In both cases the conditions of the corresponding set comprehension are violated, making no rule instances. So, as you can imagine, it's possible to write an inductive rule with zero instances! In class we've only gone down as far as one instance until now.

Hopefully this helps to demystify inductive rules and inductive definitions a bit more. Don't worry, you'll get the hang of it!

As an example derivation of an entailment relation, here is a derivation of $\emptyset \vdash \mathsf{A} \vee \bot \supset \mathsf{A}$ **true**, meaning that it is a theorem of constructive propositional logic (it can be entailed with no assumptions):

$$\dfrac{\dfrac{\dfrac{}{\{\mathsf{A} \vee \bot\} \vdash \mathsf{A} \vee \bot \textbf{ true}} \quad \dfrac{}{\{\mathsf{A} \vee \bot, \mathsf{A}\} \vdash \mathsf{A} \textbf{ true}}\,(\text{hyp}) \quad \dfrac{\dfrac{}{\{\mathsf{A} \vee \bot, \bot\} \vdash \bot \textbf{ true}}\,(\text{hyp})}{\{\mathsf{A} \vee \bot, \bot\} \vdash \mathsf{A} \textbf{ true}}\,(\bot\text{E})}{\{\mathsf{A} \vee \bot\} \vdash \mathsf{A} \textbf{ true}}\,(\vee\text{E})}{\emptyset \vdash \mathsf{A} \vee \bot \supset \mathsf{A} \textbf{ true}}\,(\Rightarrow\text{I})$$

A Formal Model of Logic From the programming-language theory perspective, all we've done above is get more practice defining the abstract syntax for a language, and then we defined a binary relation on it. However, it's worth noting that this particular language and relation is a small, formal model of the kind of reasoning that we will be doing in this class when we state and prove theorems. Compare the above formal derivation to the following informal proof, which basically proves the same thing.

Theorem 1. *If 7 is odd or 1=0 then 7 is odd.*

Proof. Suppose that 7 is odd or 1=0. Then we can proceed by cases.

Case (7 is odd). Then the conclusion follows immediately by assumption.

Case (1=0). Since that is clearly absurd, and absurdity implies anything, we immediately deduce that 7 is odd.

□

In this informal proof, we replace A with "7 is odd", and we replace $\bot$ with $1 = 0$, which is also clearly false. If you look closely at the above informal proof, you can see that it actually has the same structure as our derivation:

- The part where we say "Assume 7 is odd or 1=0" corresponds exactly to how we move $A \vee \bot$ into the set of assumptions Γ. Then we can reason assuming it is true.
- When we "proceed by cases", we are doing the equivalent of the $\vee$E rule. In each case, we assume a part of the disjunction and try to prove the conclusion.
- The first case is "immediate by assumption" which alludes to the hyp rule.
- The second one follows by absurdity, which alludes to $\bot$E.

For those of you who are not familiar with writing rigorous computer science proofs, this may help to give you a sense of what a proof should roughly look like: the structure of the proof is guided by the structure of the theorem that you are trying to prove. We will see more examples as we go along, and if helpful, we can try to make connections between our informal proofs, and the more formal models of these proofs. There may be something a little vertigo-inducing about using informal logic to reason about a formal model of logic itself. That takes some getting used to.

References

P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland, 1977.

D. van Dalen. *Logic and structure (3. ed.)*. Universitext. Springer, 1994. ISBN 978-3-540-57839-0.