Chapter 5

Reasoning about Inductive Definitions: Forward, Backward, Induction, and Recursion

Previously, we defined the small Vapid programming language. Since the language has a finite number of programs, its syntax was very easy to define: just list all the programs! In turn it was straightforward to define its evaluation function by cases, literally enumerating the results for each individual program. Finally, since the evaluator was defined by listing out the individual cases (program-result pairs), we could prove some (not particularly interesting) properties of the language and its programs.[1]

In an effort to move toward a more realistic language, we have introduced the syntax of a language of Boolean expressions, which was more complex than Vapid in that there are an infinite number of Boolean expressions. We did this using inductive definitions, which are much more expressive and sophisticated than just listing out programs. However, we must now answer the question: how do we define an evaluator for this infinite-program language, and more generally how can we prove properties of *all* programs in the language and the results of evaluating them? To answer this question, we introduce several new reasoning principles that arise quite naturally from the structure of inductive definitions.

5.1 Exploiting Derivations to Reason About the Derived

Recall the definition of the language of Boolean Expressions $t \in \textsc{Term} \subseteq \textsc{Tree}$:

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

From the above, we know that $\textsc{Term} = \{\, r \in \textsc{Tree} \mid \exists \mathcal{D}.\, \mathcal{D} :: r \in \textsc{Term} \,\}$. Let me take a moment to emphasize that the "$\in \textsc{Term}$" on the right side of the set comprehension is purely syntactic sugar. It is there to make clear to the human reader what set the derivations are describing elements of. If it weren't sugar, then this definition would be ill-founded: it would be unfortunate if we needed to already have Terms in order to define Terms.[2] Here we don't: we needed Trees r. I sometimes leave off that particular piece of sugar because it looks problematic in this context. So instead I could equivalently write
$\textsc{Term} = \{\, r \in \textsc{Tree} \mid \exists \mathcal{D}.\, \mathcal{D} :: r \,\}$.

[1] In general, these language properties are interesting, but because Vapid is so...vapid, the properties are trivial.

[2] Contrast this with the empty set and an infinite set, which we *did* bring into existence via ZFC axioms. You have to start somewhere if you hope to bootstrap all of mathematics!

Since $\textsc{Term}$ is defined using a set comprehension, we immediately know that for each element $t \in \textsc{Term}$, it is also true that $t \in \textsc{Tree}$, i.e. $\textsc{Term} \subseteq \textsc{Tree}$. Furthermore, for each $t \in \textsc{Term}$ there must be *at least* one derivation $\mathcal{D} \in \textsc{Deriv}$ such that $\mathcal{D} :: t$. In short, we appeal to the axiom of separation as it applies to our definition to justify the following reasoning principle:

Proposition 13.

$$\forall t. t \in \textsc{Term} \iff t \in \textsc{Tree} \land \exists \mathcal{D} \in \textsc{Deriv}. D :: t.$$

Proof. Consequence of the axiom schema of separation. □

We take advantage of this crisp connection between derivations $\mathcal{D}$ and $\textsc{Term}$s t to reason about the $\textsc{Term}$s by proving things about derivations.

5.1.1 Forward Reasoning

When we first introduced inductive rules for inductive definitions, the rules were given the *informal* interpretation of *if the premises are true then the conclusions follow.* However, inductive rules are just sets of rule instances, which are used to build these funky "data structures" called derivations. They are not statements in logic. However, we can make formal the connection between rules and their informal reading.

Take, for instance, the (rif) rule:

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

The following proposition and proof uses this rule (in the context of derivations) to produce a corresponding reasoning principle.

Proposition 14 (rif)**.** *$\forall t_1, t_2, t_3 \in \textsc{Term}$. $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \in \textsc{Term}$.*

Proof. Suppose $t_1, t_2, t_3 \in \textsc{Term}$. Then by the definition of $\textsc{Term}$, There is some $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in \textsc{Deriv}$ such that $\mathcal{D}_1 :: t_1$, $\mathcal{D}_2 :: t_2$, and $\mathcal{D}_3 :: t_3$. Then by (rif) we can construct a new derivation

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{t_1 \in \textsc{Term}} \quad \overset{\mathcal{D}_2}{t_2 \in \textsc{Term}} \quad \overset{\mathcal{D}_3}{t_3 \in \textsc{Term}}}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \in \textsc{Term}}$$

Since $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \in \textsc{Tree}$ and $\mathcal{D} :: \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$ it follows that $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \in \textsc{Term}$. □

Proposition 14 precisely formalizes the idea that our inductive rules enable what I'll call *forward reasoning* in terms of an inductive definition. Probably the most interesting part of the proof is that we could just *assume* three derivation trees $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ into existence without being on the hook to build them. In essence, their existence (in the universe of set theory) is a property of our definition of $\textsc{Term}$. We don't need to explicitly build them, but we can use Prop. 13 (specifically the left-to-right reading of $\Leftrightarrow$) to know that they just *have* to be out there. Then it's a simple step to build our bigger derivation and then use Prop. 13 again (this time from right to left) to know that we have a $\textsc{Term}$.

We can construct the same kind of proposition for (rtrue), but it's a fantastically boring instance of forward reasoning:

Proposition 15 (rtrue)**.** *$\mathsf{true} \in \textsc{Term}$*

Proof. Let $\mathcal{D} = \frac{}{\mathsf{true} \in \textsc{Term}}$. Then since $\mathsf{true} \in \textsc{Tree}$ and $\mathcal{D} :: \mathsf{true}$ it follows that $\mathsf{true} \in \textsc{Term}$. □

In essence this is saying "if all of the premises of (rtrue) hold, then $\mathsf{true} \in \textsc{Term}$." Since there are no premises, they are "all" vacuously true: boring case of forward reasoning!

Let's recap: in both of these propositions, what we've done is exploit the fact that $\textsc{Term}$ was inductively defined in terms of derivations (essentially a data structure) built from rules (which are just sets of rule instances) to deduce a general principle of reasoning as a logical statement (Prop. 14). The intuitive interpretation of our rules is *reflected* into actionable logical reasoning principles by proving forward-reasoning propositions.

Now in day-to-day practice, logicians and mathematicians (and computer proof assistants) don't bother *explicitly* proving the forward reasoning principles as I have done here: they take them for granted, and in a proof will simply say "by (rif), we get ..." as if they were directly appealing to the inductive rule, rather than the proposition that it reflects. Most of the time this "pun" between rules and propositions is fine, which is why I intentionally give these propositions the same name as the corresponding rule, but it can be useful to understand that rules *are not* propositions. However rules straightforwardly induce corresponding propositions.

5.1.2 Backward Reasoning

If we think of inductive rules as LEGO® blocks, and derivations as soaring structures that we build, then it is quite natural to view forward reasoning as a logical form of construction. Sometimes, however, we want to go in the opposite direction. Like my younger self, we often want to take things apart to understand how they work.[3] Inductive definitions can help us do this as well, this time exploiting the structure of *all* possible derivations, rather than a single inductive rule. For our inductive definition of TERM, this global reasoning about derivations is distilled into the following proposition about derivations:[4]

Proposition 16 (Principle of Cases on Derivations $\mathcal{D} :: r \in \textsc{Term}$)**.**
For all $\mathcal{D} \in \textsc{Deriv}$, $r \in \textsc{Tree}$, if $\mathcal{D} :: r \in \textsc{Term}$, then exactly one of the following is true:

1. $\mathcal{D} = \dfrac{}{\mathsf{true} \in \textsc{Term}}\ (rtrue)$ *(and thus $r = \mathsf{true}$);*

2. $\mathcal{D} = \dfrac{}{\mathsf{false} \in \textsc{Term}}\ (rfalse)$ *(and thus $r = \mathsf{false}$);*

3. *For some $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in \textsc{Deriv}$ and $r_1, r_2, r_3 \in \textsc{Tree}$, $\mathcal{D}_1 :: r_1$, $\mathcal{D}_2 :: r_2$, $\mathcal{D}_3 :: r_3$, and*

$$\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ r_1 \in \textsc{Term}\end{matrix} \quad \begin{matrix}\mathcal{D}_2 \\ r_2 \in \textsc{Term}\end{matrix} \quad \begin{matrix}\mathcal{D}_3 \\ r_3 \in \textsc{Term}\end{matrix}}{\mathsf{if}\, r_1\ \mathsf{then}\, r_2\ \mathsf{else}\, r_3 \in \textsc{Term}}\ (rif)$$

(and thus $r = \mathsf{if}\, r_1\ \mathsf{then}\, r_2\ \mathsf{else}\, r_3$).

Based on our understanding of inductive rules and the structure of derivations as disciplined trees of rule instances, the above statement seems in line with our intuitions. We state it explicitly for two reasons.

First, we will act as though this proposition is "given for free" from our inductive definition, this is not strictly true. Just as the forward reasoning principles in the last section were propositions that in principle had to be proven, the same is true here. I've mentioned that we can encode the idea of inductive rules and derivations directly in set theory, where an inductive rule is some kind of set, and a derivation tree is another kind of set, etc. If we were to do this, we could explicitly prove the above proposition against this representation. However, diving that deep involves too much "machine language" programming for our purposes: it would probably shed less light than heat at this point. At the least we can use something more akin to assembly language (a smidge higher-level than machine language) as our starting point, which makes life a little easier, albeit not quite as easy as we like. For this reason we will build new easier principles on top of this, but we'll know how to hand-compile our statements down to the "assembly language" level. This can be helpful (at least it has been for me) when it comes to understanding whether what you have written down actually makes mathematical sense.

Second, which is related to the first, we need to start somewhere. We need some "rules of the game" to work with. Taking this principle as given is a nice starting point in my opinion. If you'd like to see what the bottom looks like, I can point you toward some further reading. Instead, we will assume going forward that whenever you have an inductive definition, you get a corresponding principle of cases on the structure of derivations that you *could* prove if you cared to first explicitly represent derivation trees as sets.

[3] Ideally not rendering them permanently inoperable along the way, as my younger self often did, to my parents' chagrin.

[4] You are encouraged to rewrite this proposition in fully formal notation for practice. The prose makes it a bit gentler though!

Inversion Lemmas. Okay, let's use this reasoning principle for derivations to prove something (lame) about terms. Recall again our ever-delightful (rif) rule:

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

If we compare it to the other two, we might notice that this is the only rule for producing if $\textsc{Term}$s. Thus our intuition is that when given a $\textsc{Term}$ of the form $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3$, that each of its constituent $\textsc{Tree}$s is also a $\textsc{Term}$. Well, that's true and we are now equipped to *prove it*!

Proposition 17. *$\forall r_1, r_2, r_3 \in \textsc{Tree}.\ \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term} \Rightarrow r_1, r_2, r_3 \in \textsc{Term}.$*

Proof. Let $r_1, r_2, r_3 \in \textsc{Tree}$ and suppose that $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}$. Then there is some derivation $\mathcal{D}$ such that $\mathcal{D} :: \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}$. By applying Prop. 16 to it, we deduce that one of these is true:

1. $\mathcal{D} = \dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue})$ (and thus $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{true}$);
2. $\mathcal{D} = \dfrac{}{\mathsf{false} \in \textsc{Term}}\ (\text{rfalse})$ (and thus $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{false}$); or
3. For some $\mathcal{D}_a, \mathcal{D}_b, \mathcal{D}_c \in \textsc{Deriv}$ and $r_a, r_b, r_c \in \textsc{Tree}$,

$$\mathcal{D} = \frac{\overset{\mathcal{D}_a}{r_a \in \textsc{Term}} \quad \overset{\mathcal{D}_b}{r_b \in \textsc{Term}} \quad \overset{\mathcal{D}_c}{r_c \in \textsc{Term}}}{\mathsf{if}\ r_a\ \mathsf{then}\ r_b\ \mathsf{else}\ r_c \in \textsc{Term}}\ (\text{rif}) \quad (\text{and thus } \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{if}\ r_a\ \mathsf{then}\ r_b\ \mathsf{else}\ r_c).$$

An Aside: Renaming Quantifiers By applying Prop. 16 to $\mathcal{D}$, we replace $\mathcal{D}$ from the proposition with...well our current $\mathcal{D}$ which just happens to have the same name, but more importantly, we replace r from the proposition with $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3$. Another **really important** thing is that *right before* applying Prop. 16, we **rename** the set names $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, r_1, r_2, r_3$ that were quantified in case 3 to be $\mathcal{D}_a, \mathcal{D}_b, \mathcal{D}_c, r_a, r_b, r_c$. Changing the names of r_1, r_2, r_3 in the proposition is critical to avoid confusing them with the set names r_1, r_2, r_3 introduced by the proposition that we are currently trying to prove. I renamed $\mathcal{D}_1, \mathcal{D}_2$, and $\mathcal{D}_3$ just so that the subscripts of the $\mathcal{D}$s would match the rs that they are derivations of: doing that wasn't strictly necessary, but more a matter of clarity and taste. But it goes to show that you can rename quantified set names whenever you like, but sometimes doing so is necessary if you hope to produce a precise and correct proof.

Now back to our proof. It now suffices to show that each of the above cases implies $r_1, r_2, r_3 \in \textsc{Term}$. so we proceed by analyzing each case.

Case 5 (rtrue). Suppose $\mathcal{D} = \dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue})$ (and thus $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{true}$). Since $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \neq \mathsf{true}$ (i.e., $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{true} \Rightarrow \bot$), we deduce a contradiction $\bot$, from which $r_1, r_2, r_3 \in \textsc{Term}$ can be immediately deduced.

Case 6 (rfalse). Suppose $\mathcal{D} = \dfrac{}{\mathsf{false} \in \textsc{Term}}\ (\text{rfalse})$ (and thus $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{false}$). Since $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \neq \mathsf{false}$ (i.e., $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{false} \Rightarrow \bot$), we deduce a contradiction $\bot$, from which $r_1, r_2, r_3 \in \textsc{Term}$ can be immediately deduced.

Case 7 (rif). Suppose $\mathcal{D} = \dfrac{\overset{\mathcal{D}_a}{r_a \in \textsc{Term}} \quad \overset{\mathcal{D}_b}{r_b \in \textsc{Term}} \quad \overset{\mathcal{D}_c}{r_c \in \textsc{Term}}}{\mathsf{if}\ r_a\ \mathsf{then}\ r_b\ \mathsf{else}\ r_c \in \textsc{Term}}\ (\text{rif})$ for some derivations $\mathcal{D}_a, \mathcal{D}_b, \mathcal{D}_c$ and $\textsc{Tree}$s r_a, r_b, r_c. (and thus $\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 = \mathsf{if}\ r_a\ \mathsf{then}\ r_b\ \mathsf{else}\ r_c$). It follows, then, that $r_1 = r_a$, $r_2 = r_b$, and $r_3 = r_c$. Since $\mathcal{D}_a :: r_1$, $\mathcal{D}_b :: r_2$, and $\mathcal{D}_c :: r_3$, we deduce by the definition of $\textsc{Term}$ that $r_1, r_2, r_3 \in \textsc{Term}$. □

Phew! That took a lot of work! Let's take a moment to reflect a bit on the structure of this proof, which makes precise the reasoning that led us to intuitively suspect that the proposition was true even before we proved it. In essence, each $t \in \textsc{Term}$ must be justified by a derivation, the last rule of a derivation has only 3 possible shapes, and only one of them works. Thus we can analyze the last rule of the derivation to learn some new stuff. Notice though, that in order to formalize the idea that "only one of them works", we had to *explicitly* consider the obviously-not-working ones and argue formally that they don't work! In general this is a really important step that captures how you as a human almost unconsciously examine and check off the other two: "(rtrue): nope! (rfalse): nope! ...". The unexamined rule can come back to haunt you. For instance, suppose we changed the inductive definition of Term by *adding* another rule:

$$\frac{r_2 \in \textsc{Term}}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}}\ \text{(ruh-roh!)}$$

This rule roughly means that if the predicate position of the if expression is true, then we can throw whatever garbage Tree we want into the alternative branch 'cause we ain't gonna run it.[5] Then we would have new facts like $\mathsf{if\ true\ then\ false\ else\ elvis(lives!)} \in \textsc{Term}$, even though $\mathsf{elvis(lives!)} \notin \textsc{Term}$, thus breaking Prop. 17.[6] However, adding this rule would lead to one more case in Prop 16, which would lead to one more case-analysis in Prop. 17, for which we could not complete the proof: we'd b stuck. So at least we can catch the falsity of the proposition by attempting to prove it!

So you can see that backward reasoning is affected by *all* of the rules that make up your inductive definition: adding new ones or removing old ones can break your proposition in a highly "non-local" way. In contrast, forward reasoning is a property of each rule in isolation, so it can be more robust to changes in your definition.

A recap: we have demonstrated that our inductive rules give us certain *shallow* reasoning principles, that only require us to analyze one step of reasoning according to the inductive rules. This is definitely not sufficient to prove everything we would like to, but it gives us individual steps of reasoning that we can exploit within the context of more complex proofs.

Now for a terminological tidbit: a proposition like Prop. 17 is often called an *inversion lemma*[7] because the statement of the proposition reads as though you were *inverting* the meaning of the (rif) rule: if the conclusion holds then the premises hold. However, not all inversion lemmas end up corresponding to an analysis of a single rule (especially if two rules can yield the same conclusion like when we added (ruh-roh!) to our system). Nonetheless, backwards reasoning ends up being an extremely valuable resource in our arsenal of reasoning principles. It becomes even more valuable when we go beyond defining the programs in our language to defining semantics and similar artifacts.

Finally, note that the inversion lemmas corresponding to (rtrue) and (rfalse) are super boring, e.g.: $\mathsf{true} \in \textsc{Term} \Rightarrow \top$, which means roughly that if true is a term, well, then then ... well then nothing exciting to write home about (literally "truth is true"). This lemma merely confirms for we cannot extract any additional useful information from the knowledge that true is a Term. In contrast, we *were* able to learn new things about the subcomponents of an if expression.

Often, a paper will present a single proposition that it calls "the inversion lemma" which combines in a single proposition some suite of reasoning principles whose structure is guided by the inductive rules underlying a particular inductively-defined set, and whose proofs proceed by backward reasoning. This is common because backwards reasoning is really useful, so it makes sense to clearly determine and state what backward reasoning principles are nearly immediately at your disposal. The statement of that proposition can vary depending on how the author intends to reason backwards. Here is an example of such an inversion lemma for Term that is similar but different from Prop. 17: in particular, it does not *distinguish* the top-level structure of Terms, but rather treats them in full generality.

Lemma 1 (Inversion on $r \in \textsc{Term}$). *For all $r \in \textsc{Tree}$, if $r \in \textsc{Term}$ then one of the following is true:*

1. $r = \mathsf{true}$
2. $r = \mathsf{false}$
3. $r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3$ *for some* $r_1, r_2, r_3 \in \textsc{Term}$.

[5] If you think about it, some scripting languages like TCL, that incrementally parse a file while running it, and let you throw line-noise in parts that never get run, behave this way.

[6] We can prove that $\mathsf{elvis(lives!)} \notin \textsc{Term}$ (i.e., $\mathsf{elvis(lives!)} \in \textsc{Term} \Rightarrow \bot$) using exactly the same backward reasoning approach.

[7] The word "lemma" means "helper proposition, not the main thorem." They're analogous to helper functions in code.

This lemma can be proved using backward reasoning. The structure of this lemma is a bit different from Prop 17, which up-front said something about *all* $\textsc{Terms}$ r_1, r_2, r_3, whereas the third case of this lemma says *for some*. Later we'll see that this lemma can help us systematically design and implement a *parser* for $\textsc{Terms}$!

Proof. Let $r \in \textsc{Tree}$, and suppose furthermore that $r \in \textsc{Term}$.

Then it suffices to show that
$(r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3)$.

By definition of $\textsc{Term}$, we deduce that $\mathcal{D} :: r \in \textsc{Term}$ for some derivation $\mathcal{D}$. By applying Prop. 16 to $\mathcal{D}$ we deduce that either:

1. $\mathcal{D} = \dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue})$ (and thus $r = \mathsf{true}$);
2. $\mathcal{D} = \dfrac{}{\mathsf{false} \in \textsc{Term}}\ (\text{rfalse})$ (and thus $r = \mathsf{false}$); or
3. For some $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in \textsc{Deriv}$ and $r_1, r_2, r_3 \in \textsc{Tree}$,

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{r_1 \in \textsc{Term}} \quad \overset{\mathcal{D}_2}{r_2 \in \textsc{Term}} \quad \overset{\mathcal{D}_3}{r_3 \in \textsc{Term}}}{\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}}\ (\text{rif}) \quad (\text{and thus } r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3).$$

It now suffices to show that each case implies
$(r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3)$.

Case 8 (rtrue). Suppose $\mathcal{D} = \dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue})$ Then $r = \mathsf{true}$,
from which we deduce $(r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3)$.

Case 9 (rfalse). Suppose $\mathcal{D} = \dfrac{}{\mathsf{false} \in \textsc{Term}}\ (\text{rfalse})$ Then $r = \mathsf{false}$,
from which we deduce $(r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3)$.

Case 10 (rif). Suppose that for some $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in \textsc{Deriv}$ and $r_1, r_2, r_3 \in \textsc{Tree}$,

$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{r_1 \in \textsc{Term}} \quad \overset{\mathcal{D}_2}{r_2 \in \textsc{Term}} \quad \overset{\mathcal{D}_3}{r_3 \in \textsc{Term}}}{\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}}\ (\text{rif})$ Then we can deduce that for some $r_1, r_2, r_3 \in \textsc{Tree}$ $r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}$, and from the three subderivations we deduce that $r_1, r_2, r_3 \in \textsc{Term}$. This suffices to prove that $\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3$,
from which we deduce $(r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3)$.

□

Since this is one of the first proofs you've see, I'm going to walk through it again, but in painful detail, then rewrite it as you would typically see in a paper. The first is to help you understand the structure of such a proof and how it is partially guided by the structure of the formal proposition, and the second is to help you understand how most proofs in writing are really "proof sketches", which give you enough information to reconstruct the "real proof", much like like how pseudocode in a paper or textbook is meant to give you enough guidance to implement the "real algorithm" in the programming language of your choice.

Proof. To start, let me write this proposition more formally, because seeing the precise structure can help with structuring the proof:

$$\forall r \in \textsc{Tree}. r \in \textsc{Term} \Rightarrow (r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3).$$

To start, we are proving an implication. The form $\forall r \in \textsc{Term}.\Phi$ is a shorthand for $\forall r. r \in \textsc{Term} \Rightarrow \Phi$. We employ it because 99% of the time we are not writing theorems about arbitrary sets, but about elements of other sets. The standard notation is optimized for set-theorists, not PL theorists. So the proposition says **if** $r \in \textsc{Term}$, **then** case 1 holds or case 2 holds or case 3 holds. To prove a universally quantified formula $\forall r. \ldots$, we suppose that r is some arbitrary set; to prove an implication $r \in \textsc{Term}$, we prove the premise. Since $\forall r \in \textsc{Term}. \ldots$ combines both, we do both:

"Suppose that some $r \in \textsc{Tree}$, and that $r \in \textsc{Term}$."

From here it suffices to prove the consequence:
$(r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3)$.

From $r \in \textsc{Term}$ and the definition of $\textsc{Term}$, we deduce that there is some derivation $\mathcal{D}$ such that $\mathcal{D} :: r \in \textsc{Term}$. So we now consider that $\mathcal{D}$. Here, we've technically taken two steps. First, we *deduce* that $\exists D \in \textsc{Deriv}.\mathcal{D} :: r$ and simultaneously be begin to *use* that proposition by taking the (same) name $\mathcal{D}$ to refer to that particular derivation as well as the knowledge that $\mathcal{D} :: r$.

Now that we have a derivation $\mathcal{D}$, (since we've concluded that one exists!) we apply the Principle of Cases on Derivations to it to deduce that one of the following holds.

1. $\mathcal{D} = \dfrac{}{\mathsf{true} \in \textsc{Term}}\,(\text{rtrue})$ (and thus $r = \mathsf{true}$);
2. $\mathcal{D} = \dfrac{}{\mathsf{false} \in \textsc{Term}}\,(\text{rfalse})$ (and thus $r = \mathsf{false}$);
3. For some $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in \textsc{Deriv}$ and $r_1, r_2, r_3 \in \textsc{Tree}$,
$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{r_1 \in \textsc{Term}} \quad \overset{\mathcal{D}_2}{r_2 \in \textsc{Term}} \quad \overset{\mathcal{D}_3}{r_3 \in \textsc{Term}}}{\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}}\,(\text{rif}) \quad \text{(and thus } r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3\text{)}.$$

Notice that I have not renamed any quantified variables in the third case, since none of them interfere with the variables that I am currently considering. In particular, the sets r_1, r_2, r_3 in the third disjunct of our goal are existentially quantified, so we can rename them later if we need or want to.

From the Principle of Cases on Derivations we deduced that one of the three statements about $\mathcal{D}$ and r is true. It now suffices to show that each of these three cases implies $(r = \mathsf{true}) \vee (r = \mathsf{false}) \vee (\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3)$. I know: things are looking good for us, but let's be thorough and finish the job! Let's make our proof nearly computer-checkable.

So we now know that one of the above 3 things is true, and we want to show that one of the three original things up above holds: we are using one disjunction to prove another. So as with our small model of propositional logic, we use (or *eliminate*) a disjunction by separately assuming each of the three cases and trying to prove the conclusion. On the other end, we can establish (or *introduce*) a disjunction by proving *any one* of the disjuncts. We don't need to prove all of them, otherwise we'd actually be proving a conjunction. So the usual prose for using a the results of the Principle of Cases on Derivations is to say something like:

"We proceed by cases on the structure of $\mathcal{D}$"

And then write out the cases separately like the following.

Case 11 (rtrue). Suppose $\mathcal{D} = \dfrac{}{\mathsf{true} \in \textsc{Term}}\,(\text{rtrue})$ Then $r = \mathsf{true}$, so one of the three disjuncts holds.

Case 12 (rfalse). Suppose $\mathcal{D} = \dfrac{}{\mathsf{false} \in \textsc{Term}}\,(\text{rfalse})$ Then $r = \mathsf{false}$, so one of the three disjuncts holds.

Case 13 (rif). Suppose that for some $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in \textsc{Deriv}$ and $r_1, r_2, r_3 \in \textsc{Tree}$,

$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{r_1 \in \textsc{Term}} \quad \overset{\mathcal{D}_2}{r_2 \in \textsc{Term}} \quad \overset{\mathcal{D}_3}{r_3 \in \textsc{Term}}}{\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}}\,(\text{rif})$ Then we can deduce that for some $r_1, r_2, r_3 \in \textsc{Tree}$ $r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}$, and from the three subderivations we deduce that $r_1, r_2, r_3 \in \textsc{Term}$. This suffices to prove that $\exists r_1, r_2, r_3 \in \textsc{Term}.\, r = \mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3$, so one of the three disjuncts holds.

□

Notice that all the way down, the structure of the proof was analogous to the structure of proofs in our small formal model of propositional logic. What I haven't formally presented is how to introduce or eliminate $\exists$ or $\forall$ in CPL. Ideally we can avoid formalizing those, but rather get more comfortable with them through practice.

Finally, let me rewrite this proof as a proof sketch, as often appears in the literature. This mostly involves leaving out details that a seasoned human theorem-prover will be able to fill in herself.

Proof. Suppose $r \in \textsc{Term}$. We then proceed by cases on the structure of $\mathcal{D}$

Case 14 $\left(\mathcal{D} = \dfrac{}{\textsf{true} \in \textsc{Term}}\ (\text{rtrue})\right)$. Then $r = \textsf{true}$ immediately.

Case 15 $\left(\mathcal{D} = \dfrac{}{\textsf{false} \in \textsc{Term}}\ (\text{rfalse})\right)$. Analogous to the previous case.

Case 16 $\left(\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{r_1 \in \textsc{Term}} \quad \overset{\mathcal{D}_2}{r_2 \in \textsc{Term}} \quad \overset{\mathcal{D}_3}{r_3 \in \textsc{Term}}}{\textsf{if}\ r_1\ \textsf{then}\ r_2\ \textsf{else}\ r_3 \in \textsc{Term}}\ (\text{rif})\right)$. Then $r = \textsf{if}\ r_1\ \textsf{then}\ r_2\ \textsf{else}\ r_3$ immediately, and from the three subderivations we deduce that $r_1, r_2, r_3 \in \textsc{Term}$.

□

Now for some explanation. Roughly speaking, an inversion lemma is just a way of saying that if the conclusion of an inductive rule holds, then the premises of the rule hold as well. In general, things get more complex, especially because an inductive definition may have two different derivations for the same element of the defined set (e.g. from the entailment relation, $\{\,\top\,\} \vdash \top$ true: I leave it to you to find two derivations). Nonetheless, there is a corresponding notion of inversion lemmas in this case, but it may merge rules that can produce the same result. Often we won't bother proving these inversion lemmas, especially when the inductive definition in question has a quite simple structure. Then the proof could be done using backwards reasoning without incident. However, there do exist systems where the desired inversion lemmas require substantial nontrivial proofs to establish (this happens often for certain proof systems of philosophical logic, but that's a bit far afield from this class).

Nonetheless, we will often apply inversion lemmas going forward so you should know how to prove them, if only to make sure that you stated them correctly.

As mentioned earlier, these inversion lemmas are sometimes useful when thinking about implementing artifacts in code that are related to set being inductively defined. In the case of $r \in \textsc{Term}$, we get a basis for *implementing a parser* for $\textsc{Term}$s. Essentially, for our purposes, a parser is a program that given some tree r, tries to (implicitly) build a derivation $\mathcal{D} :: r \in \textsc{Term}$ starting from the bottom and working upwards. If we look at the lemmas, we can see that at each point, the next step of searching is relatively clear. When we introduce more sophisticated inductive definitions like relations that associate programs with their resulting values, or relations for specifying which programs are well-typed, we will specialize the inversion lemmas to distinguish inputs (e.g., input program) from outputs (e.g., the result of evaluation).

5.2 Proving that something is false

So far in class we have mostly been proving that something is true, for example that "There is a program in Vapid1 with undefined result."

Sometimes, we want to prove that something is *not* true though, for example, "There is no Vapid 0 program with undefined result." Proving something of the form "not P" is common, so we should make sure we understand how to do that.

Suppose I have some proposition P. I may want to prove that "P is false" or "not P." In symbolic notation, this is written

$$\neg P.$$

To prove something of this form, the standard practice is to prove that "If P is true then *absurdity* follows." In logic, we represent absurdity[8] with the symbol $\bot$, which is typically given the name "bottom." So for our purposes, $\neg P$ is just an abbreviation for $P => \bot$. The intuition is that if P is true then something is really broken in the world.

Though we haven't explicitly stated it before, there are a lot of things that we already know are not true, meaning that they imply $\bot$. For instance, we know that the atom true is not the same as the atom false. In our typical mathematical notation we write this as

$$\textsf{true} \neq \textsf{false}$$

[8] You may have heard the word "contradiction" as a synonym for absurdity. For technical reasons I'm avoiding that word, and I also want to assure you that what I am about to demonstrate is *not* "proof by contradiction."

But this is shorthand for

$$\neg(\mathsf{true} = \mathsf{false}) \quad \text{(i.e., ”it is not the case that } \mathsf{true} = \mathsf{false}\text{.”)}$$

and *that* is shorthand for

$$(\mathsf{true} = \mathsf{false}) => \bot \quad \text{(i.e., “if } \mathsf{true} = \mathsf{false} \text{ then the world is broken.”)c}$$

We can use knowledge of this proposition to prove that something is false about our language of Boolean Expressions:

Proposition 18. $\mathsf{true} \not\Downarrow \mathsf{false}$.

Rewriting this symbolically, we are proving that $\neg(\mathsf{true} \Downarrow \mathsf{false})$, i.e., that $(\mathsf{true} \Downarrow \mathsf{false}) \Rightarrow \bot$. We are proving an implication and we already know how to do that: assume the premise and use that to prove the conclusion.

Proof. Suppose that $\mathsf{true} \Downarrow \mathsf{false}$. By inversion on $t \Downarrow v$, we that for all values v, if $\mathsf{true} \Downarrow v$ then $v = \mathsf{true}$. Specializing this for our assumption, it follows that $\mathsf{false} = \mathsf{true}$. But that's absurd (i.e., we apply $(\mathsf{true} = \mathsf{false}) => \bot$ to deduce absurdity $\bot$).

Thus we've proven that it's absurd that $\mathsf{true} \Downarrow \mathsf{false}$ or rather $\mathsf{true} \Downarrow \mathsf{false} \Rightarrow \bot$. □

5.3 Inductive Reasoning

The strategy that we use to define an evaluation relation or function and prove properties about it follows our ongoing theme that *the structure of your definitions guides the structure of your reasoning*. In the case at hand, we defined the syntax of the Boolean Expressions using an *inductive definition*, which consisted of a set of *inductive rules*, whose *instances* could be used to build *derivations* that “prove” which TREEs we want to accept as TERMs. For our purposes, an inductive definition “automatically” provids reasoning principles tailored to the particular definition, just like each axiom of set theory gives us a reasoning principle that we can work with.[9]

We took the Principle of Cases on Derivations as a reasoning principle that lets us prove properties of terms based on the “shallow” structure of derivations. This principle is subsumed by a more powerful one that enables our reasoning to go “deeper”. The reasoning principle for our inductively defined set TERM follows.

Proposition 19 (Principle of Derivation Induction on $\mathcal{D} :: r \in \textsc{Term}$)**.**
Let Φ be a predicate on derivations $\mathcal{D} :: r \in \textsc{Term}$. Then $\Phi(\mathcal{D})$ holds for all derivations $\mathcal{D}$ if:

1. $\Phi\left(\dfrac{}{\mathsf{true} \in \textsc{Term}}\,(rtrue)\right)$ *holds;*
2. $\Phi\left(\dfrac{}{\mathsf{false} \in \textsc{Term}}\,(rfalse)\right)$ *holds;*
3. *For all $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in \textsc{Deriv}$, $r_1, r_2, r_3 \in \textsc{Tree}$, such that $\mathcal{D}_i :: r_i$,*
   If $\Phi\left(\begin{matrix}\mathcal{D}_1 \\ r_1 \in \textsc{Term}\end{matrix}\right)$, $\Phi\left(\begin{matrix}\mathcal{D}_2 \\ r_2 \in \textsc{Term}\end{matrix}\right)$, *and* $\Phi\left(\begin{matrix}\mathcal{D}_3 \\ r_3 \in \textsc{Term}\end{matrix}\right)$ *hold then*

   $$\Phi\left(\dfrac{\begin{matrix}\mathcal{D}_1 \\ r_1 \in \textsc{Term}\end{matrix} \quad \begin{matrix}\mathcal{D}_2 \\ r_2 \in \textsc{Term}\end{matrix} \quad \begin{matrix}\mathcal{D}_3 \\ r_3 \in \textsc{Term}\end{matrix}}{\mathit{if}\, r_1 \;\mathit{then}\, r_2 \;\mathit{else}\, r_3 \in \textsc{Term}}\,(rif)\right)$$

   holds.

Proof. For our purposes for now, this comes for free with the inductive definition of $t \in \textsc{Term}$. Later in the course will delve a bit deeper to make clear how logical predicates get drawn into this. □

[9] Technically we could prove this principle rather than take it as given, but once again that would send us further down the rabbit hole than is necessary or helpful.

First, let me explain the use of an "infix" reference to "if": near the beginning of the proposition we see the prose structure "A holds if B", or more briefly "A if B". This prose is formalized as $B \Rightarrow A$, *reversing* the two subformulae. In contrast, the prose "A only if B" is formalized as $A \Rightarrow B$, in the same direction. This is why the phrase "A if and only if B" is formalized as $A \Leftrightarrow B$ which is shorthand for "A if B and A only if B" or $(B \Rightarrow A) \wedge (A \Rightarrow B)$

Whenever we define a set using inductive rules, we get a principle of induction on the derivations from those rules. These principles all have the same general structure: assume that you have some property Φ of derivations. Then that property holds for all derivations if for *each* rule in the inductive rule, the property holds for a derivation of the conclusion if it held for the derivations of each of the premises. This can be most clearly seen in case 3. above, where the property holding for the 3 subderivations of the if derivation suffices to ensure that it holds for the whole derivation. The first two cases are a little different. Since the (rtrue) and (rfalse) rules have no premises, it's *vacuously* true that the property holds for all of the subderivations (because there aren't any).

Without delving into an actual proof of this, the intuition is this: In a sense, this theorem is a recipe for building up a proof that $P(\mathcal{D})$ holds for any particular $\mathcal{D}$: If we know that P holds for any derivation that is exactly an axiom, and we know that whenever we combine derivations $\mathcal{D}_i$ that satisfy P using some rule, we get a single tree that also satisfies P, then we can take *any* derivation, tear it apart, and prove that the leafs of the tree (at the top) satisfy P and we can systematically put the tree back together, proving at each step that the resulting piece satisfies P, until we've finally rebuilt the entire original tree and established that indeed $P(\mathcal{D})$ holds.

5.3.1 Aside: Reasoning about Function Definitions

So how do we use this principle of induction in practice? Below we will apply it to deduce facts about a function, but first let's spend some time using basic set-theoretic tools to deduce some properties of a function based on the structure of how we defined it. Then we'll see an example of using induction to prove something that we already *know* to be true intuitively, but that we must prove using the principle of induction. Note that this is the typical progression in math: as an intuitive human, we have the sense that something is true, but then we prove it formally so as to be sure. Sometimes we're surprised to discover that the thing we "knew" is false. So much for being perfect!

Anyway let's start with a simple function. Consider the following function definition, which yields the number of Boolean constants in a Term:

Definition 4.

$$\begin{aligned}
&\mathit{bools} : \textsc{Term} \to \mathbb{N}\\
\mathit{bools}(\mathsf{\mathit{true}}) &= 1\\
\mathit{bools}(\mathsf{\mathit{false}}) &= 1\\
\mathit{bools}(\mathsf{\mathit{if}}\, t_1 \ \mathsf{\mathit{then}}\ t_2 \ \mathsf{\mathit{else}}\ t_3) &= \mathit{bools}(t_1) + \mathit{bools}(t_2) + \mathit{bools}(t_3)
\end{aligned}$$

Remember that an equational function definition like the above ought be interpreted roughly as follows:

$$\text{Let } S = \left\{ F \in \textsc{Term} \to \mathbb{N} \;\middle|\; \begin{array}{r} F(\mathsf{true}) = 1 \wedge \\ F(\mathsf{false}) = 1 \wedge \\ \forall t_1, t_2, t_3 \in \textsc{Term}. F(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = F(t_1) + F(t_2) + F(t_3) \end{array} \right\}.$$

$$\text{Then } \mathit{bools} \in S \text{ and } \forall f \in \textsc{Term} \to \mathbb{N}. f \in S \Rightarrow f = \mathit{bools}.$$

We can restate it as a proposition, which makes it clearer that there is technically an obligation to prove that proposition:

$$\exists! \mathit{bools} \in \textsc{Term} \to \mathbb{N}. \quad \begin{array}{r} \mathit{bools}(\mathsf{true}) = 1 \wedge \\ \mathit{bools}(\mathsf{false}) = 1 \wedge \\ \forall t_1, t_2, t_3 \in \textsc{Term}. \mathit{bools}(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = \mathit{bools}(t_1) + \mathit{bools}(t_2) + \mathit{bools}(t_3) \end{array}$$

In short, there is a *unique* function in $\textsc{Term} \to \mathbb{N}$ that satisfies the three propositions, and we will use the name *bools* for it. Technically we are on the hook to prove that (1) there exists *at least one* function that satisfies those three equations; and (2) there exists *at most one* function that satisfies those three equations. For now, let's assume that both of these claims are true. Below, we will prove that a particular *scheme* (i.e., template) for equations is guaranteed to define a unique function, and that the equations for *bools* fit that scheme. We call that scheme the Principle of Recursion for $t \in \textsc{Term}$.

Anyway, we have a function now and we'll assume it exists. What do we know about it *right now*? Well we know by the axiom of separation that (1) it assigns a unique natural number to each term t, and (2) that these assignments are constrained by the equations given above. In fact, since this is a function *definition*, we know that these constraints are sufficient to uniquely describe one function.

First, let's prove something we already know about how *bools* treats one particular term.

Proposition 20. $bools(true) = 1$.

Proof. *bools* satisfies three equations, and conveniently the first one immediately completes the proof. □

Not a particularly complex proof, but a proof nonetheless! Remember: *bools* is technically just a particular subset of $\textsc{Term} \times \mathbb{N}$, in fact an infinite subset, but nonetheless we can use the few facts that we know about it to deduce facts. Here we used *equational reasoning* to deduce new facts about *bools*. It's a bit absurd for me to write the above as a proposition and a proof: no one does that in real life, they just write out the calculation as in the following example:

$$\begin{aligned}
&bools(\mathsf{if\ false\ then\ true\ else\ true})\\
=&bools(\mathsf{false}) + bools(\mathsf{true}) + bools(\mathsf{true})\\
=&1 + 1 + 1\\
=&3.
\end{aligned}$$

Each step of the above appeals to equational reasoning to learn something. In the above example, we used our sparse knowledge from the definition to deduce *additional* facts about the *bools* function. You may be rolling your eyes a bit, since you have been doing stuff like this since secondary school, but it's useful to recognize this as an instance of *deduction*: learning new facts from old. This is a proof of a theorem (that the first term equals the last). All too often we think of a mathematical function as a machine that performs calculations given inputs. Hogwash! Our mathematical function is just a table of mappings from $\textsc{Term}$ to $\mathbb{N}$. It sits there like a dead fish. Not only that, it's infinite, so we can't hope to just read down the list of entries to find the one that we want (I'm pretty sure it wouldn't fit on disk anyway, let alone in memory). Instead it is *we*, the logical deduction engines, that use *only* the fact that the function exists, and the scant few other properties that we found sufficient to uniquely *describe* the function, in particular the equational constraints, to *deduce* some of the entries in this infinite table of pairs. In short, functions don't compute, they are just sets: *calculation* is the process of deducing facts about a function. That's what we teach computers to do for us: deduce.

Okay, enough with the philosophy for now. Let's get back to deducing facts about *bools*. In the last two examples, we used equational reasoning to deduce facts about particular entries. Now, let's deduce facts about entire *classes* of entries.

Proposition 21. $\forall t \in \textsc{Term}.bools(\mathsf{if}\, t\ \mathsf{then}\ t\ \mathsf{else}\ t) = 3 * bools(t)$.

Proof.

$$\begin{aligned}
&bools(\mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t)\\
=&bools(t) + bools(t) + bools(t)\\
=&3 * bools(t).
\end{aligned}$$

□

This proposition is *not* that much different looking from the last one, but it's way more general: the last one told us a fact about *one* term, while this one tells us something about *an infinite number of terms*! So exciting. In practice, if I were on the hook to implement *bools* as a computer program (i.e. a deduction engine for this function), I could possibly exploit this fact to add an optimization to my deduction engine: "if you see a case like this, don't bother computing $bools(t)$ three times: just do it once and multiply." As programmers, we make these steps of deduction all the time while we work, at least if we are concerned about performance. In essence, this is what an optimizing compiler (or interpreter!) for a programming language does too. Here we make such a deduction explicit and justify it with a formal proof of its correctness.

5.3.2 Your First Proof By Induction

In the last segment, we showed that in the set-theoretic world, an equational function definition is just a "constraint filter" on the set of functions with a domain, winnowing it down until there can be only one.[10] Note that I am specifically saying *equational* function definition, because there are other ways to define functions (i.e., pick out an individual element of the set of functions). And we showed that we can use those equations to deduce new properties of the function...these deductions are exactly the calculations that we perform to determine the value of a function for a particular input. But we also show that those same deductions can be used to determine facts about *entire* classes of values, and given such a deduction we can "optimize" future deductions. We'll find out later that people write computer programs that automatically perform these kinds of deductions too. For instance, the subfield of program analysis called *symbolic execution* Baldoni et al. [2018] is precisely about writing an automatic and efficient "proof engine" for deducing useful facts about abstract expressions, where *symbols* play the role of our metavariables. The workaday programmer is much more comfortable with writing such deduction engines for the value of a function when given a concrete input! Shockingly enough, we call such deduction engines...wait for it..."functions."

Hopefully these side-commentaries about deduction might help you make a useful observation: many computer programs that you and others are writing can be viewed as *automated theorem provers* for very limited classes of theorems. In the case of implementing *bools* as a programming language function (which I sometimes call a *procedure* to disambiguate), the corresponding program accepts a term t, uses deduction like above to prove the specialized theorem $\exists n \in \mathbb{N}.\, bools(t) = n$, *throws away the proof*, and just gives you the number n.

Time for a theorem! Now it's worth noticing something interesting about our *bools* function if only intuitively: The function is defined to evaluate to *natural numbers* $\mathbb{N}$, but not every natural number has some term to which *bools* maps it. In particular, there is no Term such that $bools(t) = 0$! How do we know? Well, just by staring at the equations and saying something like "look, the Boolean constant cases yield 1 and the if case uses +...so it *just can't* produce 0...right?" But how do we *formally* prove that this hand-wavy explanation is correct? Clearly we can *test* this hypothesis by coming up with a bunch of concrete Terms and deducing their values and then gaining confidence in our observation. But since we have an infinite number of Terms, we cannot exhaustively test the *bools* function. Another way to say this last sentence is: we cannot appeal to the reasoning principle that we get by giving an extensional definition of a finite set like $\{ a, b, c \}$ because Term is infinite, so we *can't* give an extensional definition, so we don't get extensional reasoning.

So what to do? Use the Principle of Derivation Induction on $\mathcal{D} :: r \in \text{Term}$! And you'll find yourself using induction principles time and time again in this course.

Let's work through this proof, in more painful of detail than you would see in the literature. It's more important to understand what is really going on first, then see how people take shortcuts when writing it down (think "code" versus "pseudocode").

Proposition 22. *$bools(t) > 0$ for all $t \in$ Term.*

Proof. A note about the proof statement. Mathematicians sometimes put the quantifiers (for all, for some) at the *end* of the statement, even though it appears at the beginning of a formal presentation: $\forall t \in \text{Term}.bools(t) > 0$. Other times they leave off the quantifiers, and then you have to guess what they mean. Both of these are meant to emphasize the most important part of the statement, since a person can usually figure out the proper quantification from context. Sadly, this doesn't always work: sometimes you are left scratching your head wondering exactly what they mean, and if the proof isn't there for you to inspect, you may end up sending an annoyed email to the author to find out what the heck they meant...sadly

[10] No swords or immortals necessary!

I've been that author some times. Don't be that author. We will tend to write propositions in full formal style, though occasionally we will mix in less formal prose in the traditional style, often to make a statement (especially induction principles) easier to read.

We are going to use the Principle of Induction, but that will not get us all the way to exactly the statement above, but it will get us most of the way there.

To use the principle, we first need to pick a particular *property* Φ of derivations. Use this one:

$$\Phi(\mathcal{D}) \equiv \forall t \in \textsc{Term}. \mathcal{D} :: t \Rightarrow \mathit{bools}(t) > 0.$$

I'm justified *technically* to use $t \in \textsc{Term}$ here because every $\textsc{Term}$ is a $\textsc{Tree}$, so the property is at least well-formed: it's always reasonable to ask if $\mathcal{D} :: t$ since $:: \subseteq \textsc{Deriv} \times \textsc{Tree}$. Furthermore, I'm justified *pragmatically* to restrict my property to $\textsc{Term}$s t, in the sense that this will ultimately work out, because we already know by definition that if $t \in \textsc{Term}$ then $\mathcal{D} :: t$ for *some* derivation $\mathcal{D}$, so if I can prove this property for all *derivations*, then it will imply the property I really care about for all *terms*. We'll see this reified in the last step of our proof.

Now, specialize the conditions on the Principle of Induction according to this property, which yields 3 *lemmas*, minor propositions, to prove.

The first condition was $\Phi\left(\dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue})\right)$, so plugging in our property yields the following condition:

Lemma 2.

$$\forall t \in \textsc{Term}. \left(\dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\mathit{rtrue})\right) :: t \Rightarrow \mathit{bools}(t) > 0.$$

Proof. Suppose $t \in \textsc{Term}$, and $\left(\dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue})\right) :: t$. Then $t = \mathsf{true}$, and $\mathit{bools}(t) = \mathit{bools}(\mathsf{true}) = 1 > 0$. □

Okay, let's all acknowledge that I'm being rather pedantic in this proof. But the reason I'm doing so is just to show that there's no magic: we learned in the last subsection how to use equational reasoning to prove things, by plug-and-chug, and similarly here I just plugged my property into the definition verbatim and proved the proposition that it gave me. The annoying part is...why did I have to deal with this "for all t's that the derivation could possibly be a derivation of"? Well, because the general form of the property has to quantify over t, and only after we pick a particular derivation can we show that t must be exactly the same thing as the conclusion. Note that if we treated :: as a function from derivations to trees (which it is: $\mathcal{D} :: t$ would be the same as saying $::(\mathcal{D}) = t$), then I would still end up deducing that $\mathsf{true} = t$. In any case, all of that is detail, but ideally you see that we're being very systematic, much like a computer program would have to be. That rigour becomes helpful (and not merely tedious) only when the objects and proofs get more complicated, as well as when the prover or proof checker becomes a computer program.

Okay, I've now got two more lemmas to prove:

Lemma 3.

$$\forall t \in \textsc{Term}. \left(\dfrac{}{\mathsf{false} \in \textsc{Term}}\ (\mathit{rfalse})\right) :: t \Rightarrow \mathit{bools}(t) > 0.$$

Proof.

Suppose $t \in \textsc{Term}$, and $\left(\dfrac{}{\mathsf{false} \in \textsc{Term}}\ (\text{rfalse})\right) :: t$. Then $t = \mathsf{false}$, and $\mathit{bools}(t) = \mathit{bools}(\mathsf{false}) = 1 > 0$. □

Okay, let's be honest: I wrote the above proof by copying the previous one and replacing the trues with falses. That is to say, this proof follows essentially the same argument as the previous one. In a paper or tech report, you are more likely to see:

Proof. Analogous to the lemma for true. □

For communicating with experts this is a *good thing*: it tells me in a compressed form what the proof for this case is like, so I don't have to read through the details and discover that the proof is indeed analogous. However, it had better be true that the proof goes analogously! That is to say, better to have proven it, discover that it is analogous, and then compress the proof afterwards, rather than assume that it's analogous and be *wrong!* This is one of the pitfalls that people fall into when proving theorems. For at least the first part of the class, I will ask you to present each proof in full, so that you get more practice with proving each of the individual cases (even if "practice" means getting your cut-and-paste-and-edit right...).

Now for the last case, which is the most interesting one because it really demonstrates the power of induction:

Lemma 4. *For all $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in$ Deriv, $r_1, r_2, r_3 \in$ Tree, such that $\mathcal{D}_i :: r_i$,*
If

$$\forall t \in \textsc{Term}. \left(\begin{array}{c} \mathcal{D}_1 \\ r_1 \in \textsc{Term} \end{array} \right) :: t \Rightarrow \mathit{bools}(t) > 0,$$

and

$$\forall t \in \textsc{Term}. \left(\begin{array}{c} \mathcal{D}_2 \\ r_2 \in \textsc{Term} \end{array} \right) :: t \Rightarrow \mathit{bools}(t) > 0,$$

and

$$\forall t \in \textsc{Term}. \left(\begin{array}{c} \mathcal{D}_3 \\ r_3 \in \textsc{Term} \end{array} \right) :: t \Rightarrow \mathit{bools}(t) > 0$$

then

$$\forall t \in \textsc{Term}. \left(\dfrac{\begin{array}{c} \mathcal{D}_1 \\ r_1 \in \textsc{Term} \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ r_2 \in \textsc{Term} \end{array} \quad \begin{array}{c} \mathcal{D}_3 \\ r_3 \in \textsc{Term} \end{array}}{\mathsf{if}\, r_1\ \mathsf{then}\, r_2\ \mathsf{else}\, r_3 \in \textsc{Term}v}\ (\mathit{rif}) \right) :: t \Rightarrow \mathit{bools}(t) > 0.$$

Wow, what a mouthful! But notice the structure of the argument: all we have to prove is that if the property holds for derivations $\mathcal{D}_i$ of the subtrees r_i , then we can build a proof that the property holds for the derivation that you get when you hook together the $\mathcal{D}_i$ derivations to form a proof $\mathcal{D}$ that if r_1 then r_2 else $r_3 \in$ Term. Also, notice that the derivations $\mathcal{D}_i$ and r_i are quantified universally at the beginning. Then every reference to those names in the lemma refers to the same derivation. In contrast, each individual precondition has its own t that is quantified universally, so we can use these preconditions to deduce facts about a variety of t's. It turns out that for this proof we will use each precondition only once. Okay let's prove it!

Proof. Suppose $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \in$ Deriv, and $r_1, r_2, r_3 \in \mathit{Tree}$, and $\mathcal{D}_i :: r_i \in$ Term. Furthermore, suppose

$$\forall t \in \textsc{Term}. \left(\begin{array}{c} \mathcal{D}_i \\ r_i \in \textsc{Term} \end{array} \right) :: t \Rightarrow \mathit{bools}(t) > 0$$

holds for each $\mathcal{D}_i :: r_i$.

Now let

$$\mathcal{D} = \left(\dfrac{\begin{array}{c} \mathcal{D}_1 \\ r_1 \in \textsc{Term} \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ r_2 \in \textsc{Term} \end{array} \quad \begin{array}{c} \mathcal{D}_3 \\ r_3 \in \textsc{Term} \end{array}}{\mathsf{if}\, r_1\ \mathsf{then}\, r_2\ \mathsf{else}\, r_3 \in \textsc{Term}}\ (\text{rif}) \right).$$

It suffices to show that

$$\forall t \in \textsc{Term}. \left(\begin{array}{c} \mathcal{D} \\ \mathsf{if}\, r_1\ \mathsf{then}\, r_2\ \mathsf{else}\, r_3 \in \textsc{Term} \in \textsc{Term} \end{array} \right) :: t \Rightarrow \mathit{bools}(t) > 0,$$

so let's do it:

Suppose $t \in$ Term and $\mathcal{D} :: t$. Then $t =$ if r_1 then r_2 else $r_3 \in$ Term. Using our assumptions we can prove that $\mathit{bools}(r_i) > 0$.[11] Let me do it for one case. Apply the assumption

$$\forall t \in \textsc{Term}. \left(\begin{array}{c} \mathcal{D}_1 \\ r_1 \in \textsc{Term} \end{array} \right) :: t \Rightarrow \mathit{bools}(t) > 0,$$

[11]2) "Using our assumptions" corresponds to how you often see a proof say "by the induction hypothesis"! The induction hypotheses are just the assumptions we made that Φ applies to each subderivation-tree pair $\mathcal{D}_i, r_i$.

to the term r_1 to get:

$$\left(\genfrac{}{}{0pt}{}{\mathcal{D}_1}{r_1 \in \textsc{Term}}\right) :: r_1 \Rightarrow bools(r_1) > 0.$$

By assumption, $\mathcal{D}_1 :: r_1$, which when applied to the above, gives us $bools(r_1) > 0$.[12] We repeat this reasoning for r_2 and r_3 to deduce $bools(r_2) > 0$. and $bools(r_3) > 0$.

From there, we can calculate

$$bools(\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3) = bools(r_1) + bools(r_2) + bools(r_3).$$

But using our knowledge that $bools(r_i) > 0$, and summing up all three inequations gives
$bools(r_1) + bools(r_2) + bools(r_3) > 0 + 0 + 0 = 0$. □

Cool, so so far, we've proven three interesting lemmas, two of which are about concrete derivations (rtrue) and (rfalse), and one that is about what happens if you build a derivation using (rif) from three pre-existing derivations that satisfy our property Φ. Are we done? Technically no! Now we *apply* the principle of induction on derivations to our three lemmas to get:

Lemma 5.

$$\forall \mathcal{D} \in \textsc{Deriv}.\forall t \in \textsc{Term}.\mathcal{D} :: t \Rightarrow bools(t) > 0.$$

Great, so now we know something about *all derivations* $\mathcal{D} \in \textsc{Deriv}$. Surely we're done? The pedant says no! Now, much like the induction hypotheses from earlier, we deduce from this fact *and* the definition of Term the knowledge that we really want.

Proposition 23. $\forall t \in \textsc{Term}.bools(t) > 0$.

Proof. Suppose $t \in \textsc{Term}$. Then by the definition of Term, there exists some derivation $\mathcal{D} \in \textsc{Deriv}$ such that $\mathcal{D} :: t$. Applying Lemma 5 to that derivation, we get that $\forall t' \in \textsc{Term}.\mathcal{D} :: t' \Rightarrow bools(t') > 0$.[13] Well, I know that $t \in \textsc{Term}$, so if I apply the above proposition to it, I get that $(\mathcal{D}) :: t \Rightarrow bools(t) > 0$. And now, we already knew that $\mathcal{D} :: t$, so we apply the above proposition to this fact and we get—pant pant pant—*finally*, $bools(t) > 0$. Woohoo! □

□

Okay, I walked through this proof in *painstaking detail* since this is the first time that we are doing a proof by induction. The main point to takeaway is that there is no magic...or in a sense, the only magic that we evoked is the Principle of Induction, which I stated without proof. Somehow it turns three pretty benign lemmas into a fact about all derivations! Then we applied the definition of Term, and some basic reasoning by applying "foralls" and "if-thens" to relevant premises to get what we wanted. Now, most such proofs, when written by professional mathematicians are written in *much less detail.* By the time you are a professional, you don't want to dig through all of the details, you just want a sketch of the argument, and if some madman offered you your very own Maserati to write the full proof, you could fill in the details and drive away in a blaze of glory and the caustic smell of burnt rubber.

However, if you are a *digital computer*, in particular a mechanical proof checker, then you need way more detail than a professional mathematician to verify that a proof is true. A *theorem prover* or *mechanical proof assistant* can surely fill in some of these details (just like our functional program that automatically generates proofs of facts), but to be rock-solid sure that a theorem is true, there should be a pedantic *proof checker* hiding somewhere in your tool that takes the proof and simply checks that every last detail is there. This is how tools like the Coq[14] proof assistant work: under the hood somewhere is the pedantic proof, waiting to be checked by a very simple, and pedantic, proof checker.

Later I'll show you what a mathematician's proof (which you can think of as pseudocode for a *real* proof, much like pseudocode for a real program) looks like. Those are nice for communicating with humans, but first you want to make sure that when push comes to shove, you can write the real thing, and understand that much of it is super-mechanical, and furthermore can be mechanically checked.

[12] Notice the terminology, that I'm "applying" a proposition to a "term"...sounds a lot like I'm applying a function to an argument, eh? This is no coincidence!

[13] Notice that without fanfare, I renamed the quantified t in the result to t' so that it doesn't cause us problems in a second. You can always rename quantified names, and sometimes it helps avoid confusion.

[14] `https://coq.inria.fr/`

5.3.3 Rule Induction

Above we used induction to prove a universal fact about a particular function, exploiting its equational definition. Now we consider another *proof principle*, which can streamline some other proofs that we will write.

Returning to the problems we set out at the start, we are interested in defining an evaluator over Terms, which implies being able to reason about the Terms in our language. However, we've inherited a principle for reasoning about *derivations*, not Terms. Working with the derivations seems a bit indirect: notice all the work that we had to do above to get from a theorem about derivations to a theorem about Terms. However, since we defined the set of Terms using the derivations, one would expect that we could use this principle to prove properties of Terms. Rather than fiddle with derivations every time we want to talk about Terms, we can introduce and apply a principle that lets us reason about properties of Terms directly.

Proposition 24 (Principle of Rule Induction for $r \in \text{Term}$)**.**
Let P be a predicate on Trees r. Then $P(r)$ holds for all Terms t if:

1. *$P(\mathsf{true})$ holds;*
2. *$P(\mathsf{false})$ holds;*
3. *For all $r_1, r_2, r_3 \in \text{Tree}$, if $P(r_1)$ $P(r_2)$, and $P(r_3)$ hold then $P(\mathsf{if}\, r_1\, \mathsf{then}\, r_2\, \mathsf{else}\, r_3)$ holds.*

As we'll see below, rule induction is a nice tool to have, especially when reasoning about *abstract syntax*, like the set Term, but can also be used for other inductively defined sets. The reason it is called *rule* induction is that the structure of each induction lemma closely mirrors one of the inductive rules used to form Term, without mention of derivations. This principle is phrased in terms of a property $P(r)$ for $r \in \text{Tree}$ rather than $t \in \text{Term}$. This difference is not critical: the variant that considers properties of Terms is also true (later in the course we will show a straightforward way to instantiate that). Sometimes, one desires to prove a property $Q(t)$ that is most conveniently phrased such that we can assume that the argument is a term. To do so, simply use the property $P(r) \equiv r \in \text{Term} \wedge Q(r)$. This form of proposition is particularly helpful for the third case, where one can now assume that r_1, r_2, r_3 are Terms while proving the property for the conditional expression.

A final note. Consider again Prop. 16, the principle of cases on derivations. We can *prove* it by induction on derivations $\mathcal{D}$, using Prop. 19! One curious aspect of that proof is that we *never* need to make use of the "induction hypotheses" to do so. This aspect of that proof is what makes reasoning by cases a "shallow" analysis of derivation, whereas reasoning by induction in general performs a "deep" analysis of derivations. In this sense, proof by cases on derivations is a degenerate variant of proof by induction on derivations: anything that you can prove by cases, you can also prove by induction. But when given a glass ceiling that you'd like to break, why use a concussion grenade when a ball peen hammer will do?

$\mathit{bools}(t) > 0$ **revisited** Okay, now that we have a new inductive tool, how shall we use it? To demonstrate, I will *re-prove* Prop. 22, but this time by using rule induction on $t \in \text{Term}$ instead of induction on derivations. Note how similar these two proofs are in structure.

Proposition 25. $\forall t \in \text{Term}.\mathit{bools}(t) > 0$

Proof. By rule induction for $t \in \text{Term}$.

The property for this proof is a little simpler than for our previous rendition:

$$\Phi(t) \equiv \mathit{bools}(t) > 0.$$

In fact, we just stripped the quantifier for t, and treated the remainder as the property.

Now for our induction lemmas and their proofs

Lemma 6. *$\mathit{bools}(\mathsf{true}) > 0$.*

Proof. $\mathit{bools}(\mathsf{true}) = 1 > 0$ □

Lemma 7. *$\mathit{bools}(\mathsf{false}) > 0$.*

Proof. $bools(\mathsf{false}) = 1 > 0$ □

Lemma 8.
$\forall t_1, t_2, t_3 \in \textsc{Term}.\, bools(t_1) > 0 \wedge bools(t_2) > 0 \wedge bools(t_3) > 0 \Rightarrow bools(\textit{if}\ t_1\ \textit{then}\ t_2\ \textit{else}\ t_3) > 0.$

Proof. Suppose $t_1, t_2, t_3 \in \textsc{Term}$, $bools(t_1) > 0$, $bools(t_2) > 0$, and $bools(t_3) > 0$. Then

$$bools(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = bools(t_1) + bools(t_2) + bools(t_3) > 0 + 0 + 0 = 0.$$

□

□

Example: Rule Induction for Propositional Entailment To give you another example of a rule induction principle, as we are calling it, here is the statement of the Principle of Rule Induction for $\Gamma \vdash p\ \mathsf{true}$ (i.e. elements $\langle \Gamma, p \rangle \in \cdot \vdash \cdot\ \mathsf{true}$). It can be proven in terms of the Principle of Induction on Derivations of $\mathcal{D} :: \Gamma \vdash p\ \mathsf{true}$, which we leave as an exercise for you to state. For reference, here is the inductive definition:

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

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

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

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

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

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

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

The corresponding principle of rule induction follows:

Proposition 26 (Principle of Rule Induction for $\Gamma \vdash p\ \mathsf{true}$)**.** *Let Φ be a property of entailments $\Gamma \times \textsc{Prop}$. Then $\Phi(\Gamma, p)$ holds for all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p \in \textsc{Prop}$ such that $\Gamma \vdash p\ \mathsf{true}$ if*

1. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p \in \textsc{Prop}$, if $p \in \Gamma$ then $\Phi(\Gamma, p)$.*
2. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $\Phi(\Gamma, \top)$.*
3. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, if $\Phi(\Gamma, \bot)$ then $\Phi(\Gamma, p)$.*
4. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2 \in \textsc{Prop}$, if $\Phi(\Gamma, p_1)$ and $\Phi(\Gamma, p_2)$ then $\Phi(\Gamma, p_1 \wedge p_2)$.*
5. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2 \in \textsc{Prop}$, if $\Phi(\Gamma, p_1 \wedge p_2)$ then $\Phi(\Gamma, p_1)$.*
6. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2 \in \textsc{Prop}$, if $\Phi(\Gamma, p_1 \wedge p_2)$ then $\Phi(\Gamma, p_2)$.*
7. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2 \in \textsc{Prop}$, if $\Phi(\Gamma, p_1)$ then $\Phi(\Gamma, p_1 \vee p_2)$.*
8. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2 \in \textsc{Prop}$, if $\Phi(\Gamma, p_2)$ then $\Phi(\Gamma, p_1 \vee p_2)$.*
9. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2, p_3 \in \textsc{Prop}$, if $\Phi(\Gamma \cup \{\, p_1 \,\}, p_3)$ and $\Phi(\Gamma \cup \{\, p_2 \,\}, p_3)$ then $\Phi(\Gamma, p_3)$.*
10. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2 \in \textsc{Prop}$, if $\Phi(\Gamma \cup \{\, p_1 \,\}, p_2)$ then $\Phi(\Gamma, p_1 \supset p_2)$.*
11. *For all $\Gamma \in \mathcal{P}(\textsc{Prop})$, $p_1, p_2 \in \textsc{Prop}$, if $\Phi(\Gamma, p_1 \supset p_2)$ and $\Phi(\Gamma, p_1)$ then $\Phi(\Gamma, p_2)$.*

Handling Side Conditions in Induction Principles Notice how in the first clause, which corresponds to the (hyp) rule for CPL, that the side-condition $p \in \Gamma$ becomes a premise of the lemma. However, there is no premise of the form $\Phi(p \in \Gamma)$: that doesn't even make sense really. However, the rest of the clauses have premises of the form $\Phi(\Gamma \vdash p \ \texttt{true})$ corresponding to premises of the inductive rules.[15] This is another way in which side-conditions differ from premises in inductive definitions: side-conditions do not induce an "induction hypothesis", just a plain ole' hypothesis!

One side-note. Often, the principle of rule induction for a set that represents abstract syntax is called a principle of *structural induction* for that set (e.g. structural induction for $t \in \textsc{Term}$), because the inductive rules used to justify the principle exactly mirror the syntactic structure of the set's elements. For example, the (rif) rule for Term has three premises, one for each of its subterm. So each rule describes a structural formation rule for this kind of abstract syntax. In contrast, one would *not* call rule induction on $\Gamma \vdash p \ \texttt{true}$ "structural induction" since the rules do not mirror our intuitive conception of the structure of the set's elements. Rule induction can feel somewhat awkward when the derivations don't mirror the structure of the elements. Being able to reference derivation trees (and the side-conditions of the last rule used) in context can really help the reader, and prover, clarify the connection between the premises and the conclusions.

Historically, the concepts and terminology of structural induction, rule induction, and induction on derivations were developed independently, but we see that they can all be viewed as variations on a common theme: inductive rules define sets *and* lead to (inductive) reasoning principles.

I introduce this principle of induction primarily to show that we can easily get these principles, induction on derivations, and induction on elements, for *any* inductively defined set, regardless of what kind of set you are defining: syntax, relations, functions, or what have you. Whether you will find the principle useful or not is a different story.

5.4 Defining Functions

We've now developed a powerful tool, induction, that we can use to prove properties of Terms, but we have little experience using it yet. On another topic, we still need some way to justify equational definitions of functions on Terms. Let's kill two birds with one stone: we will use Proposition 86 to prove a new principle: that we *always* describe a unique function if we require it to satisfy a particular *scheme* of equations that in general refers to the name of that function on both sides of the equal sign. This property, that the function name *recurs* on both sides of the characterizing equations, is precisely what makes for a *recursive* function definition.

We'll use the resulting principle to produce, with no remaining proof obligations, our first non-trivial function over an inductively-defined infinite set. As with any equational function definition, we'll see that the structure of these function definitions will allow us to reason about their properties. For instance, we can calculate equationally such a function maps particular inputs to outputs.

Enough lead-up: let's state the principle:

Proposition 27 (Principle of Definition by Recursion for $t \in \textsc{Term}$)**.** *Let S be some set and $s_t, s_f \in S$ be two of its elements and*

$$H_{\mathit{if}} : S \times S \times S \to S$$

be a function on S. Then there exists a unique function

$$F : \textsc{Term} \to S$$

such that

1. $F(\mathsf{true}) = s_t$;
2. $F(\mathsf{false}) = s_f$;
3. $F(\mathsf{if}\, t_1 \ \mathsf{then}\, t_2 \ \mathsf{else}\, t_3) = H_{\mathit{if}}(F(t_1), F(t_2), F(t_3))$.

[15] Take care to distinguish between premises *of inductive rules* and premises *of an implication*. Unfortunately these different things have the same name because they are somewhat analogous.

Before we start using the principle, let's talk a bit about its structure to give you some intuition for why this makes sense. This principle is conceptually about mapping Terms to Ss, tree-node by tree-node. To get a sense of this, consider the specific Term if false then true else if true then false else true. Given some set S, elements s_t, s_f and function H_{if}, the function F induced by them could be shown via equational reasoning to satisfy the following equation:

$$F(\textsf{if false then true else if true then false else true}) = H_{if}(s_f, s_t, H_{if}(s_t, s_f, s_t))$$

To get from the Term on the left to the expression on the right, we "simply" replaced all instances of true with s_t, false with s_f, and if with H_{if}, treating then and else as placeholders for commas. The equations give a "single-step" description of this larger structure. In fact, you are encouraged to write down a few terms and use the equations to demonstrate this correspondence. Understanding this correspondence can help you think about how to craft the function definition that you really want.

This principle can be proved using what we've already learned about inductive definitions and their associated induction principles. You'll get to see this in action on your homework.

Now, let's use Proposition 115 to define a function! According to the proposition all we need is:

1. some set (S);
2. two elements (s_t and s_f) of that set, though you can use the same element for both; and
3. some function (H_{if}) from any three elements of that set to a fourth.

For our example, I'll pick:

1. the set of natural numbers: $S = \mathbb{N}$.
2. the number 1 for s_t, and 0 for s_f: $s_t = 1$ and $s_f = 0$.
3. and the function $H(n_1, n_2, n_3) = n_1 + n_2 + n_3$ which just sums up all the numbers: $H : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Well, then according to Proposition 86, there is a *unique* function F : Term $\to \mathbb{N}$ with the properties that:

$$F(\textsf{true}) = 1; \tag{5.1}$$
$$F(\textsf{false}) = 0; \text{ and} \tag{5.2}$$
$$F(\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3) = H(F(t_1), F(t_2), F(t_3)) = F(t_1) + F(t_2) + F(t_3). \tag{5.3}$$

That means that these three equations (properties) *uniquely* characterize some function from Terms to natural numbers. At this point all we know is that there *is* a function that satisfies these properties, and that there's only one. This isn't much: all we understand about this *black box* of a function is the equations that it satisfies. But broadly speaking, what does this function really mean? We could learn some things about this function by using the equations to calculate what the function maps certain terms to, but I'll save that exercise for homework. For now, I hope you'll trust me that this particular function associates every Termin our language to the number of trues that appear in it. Thus it's reasonable to call this function *trues*.

What I've shown here is a rather *longhand* way of writing down a function definition: we take the Principle of Recursion at its word literally, choose the necessary components, and then conclude that there's some function that satisfies the set of equations that you get after you specialize the proposition for your particular choices (as I've done above). In textbooks and papers, writers rarely show things in this much painful detail. Instead, they cut to the chase and simply give the equations that you get at the end. We'll call that the *shorthand* way of defining a function by recursion.[16]

Considering our example again, here is the typical shorthand definition of the same function:

[16] This *shorthand* and *longhand* terminology is my own creation. I don't think you'll find it in the literature.

Definition 5.

$$\begin{aligned} \mathit{trues} &: \textsc{Term} \to \mathbb{N} \\ \mathit{trues}(\mathsf{true}) &= 1 \\ \mathit{trues}(\mathsf{false}) &= 0 \\ \mathit{trues}(\mathsf{if}\, t_1 \ \mathsf{then}\ t_2 \ \mathsf{else}\, t_3) &= \mathit{trues}(t_1) + \mathit{trues}(t_2) + \mathit{trues}(t_3). \end{aligned}$$

The function is described by the final specialized version of the equations, and it's up to you (the reader) to figure out which S, s_t, s_f, and H_{if} give you these equations …which is often not hard.

The shorthand definition above can be read this way: ignoring the name we're giving the function (*trues*), and simply taking the three equations, we are saying that:

$$\mathit{trues} \in \{\, F \in \textsc{Term} \to \mathbb{N} \mid P(F) \,\}$$

Where $P(F)$ is the combination of equations (5.1)-(5.3) above, with leading universal quantifers placed appropriately. The important thing is that in order for $P(F)$ to be a good definition, i.e., a *definite description*, the set we define above should have *exactly* one element, which means that *trues* is that element.

Why am I beating this to death? Because it's easy to write a so-called "definition" with equations that's not a definition at all![17] Let's consider two simple examples. Take the natural numbers $\mathbb{N}$, and suppose I claim I'm defining a function $F : \mathbb{N} \to \mathbb{N}$ by the equation $F(n) = F(n)$. Well that doesn't define a *unique* function at all because

$$\{\, F \in \mathbb{N} \to \mathbb{N} \mid \forall n \in N.F(n) = F(n) \,\} = \mathbb{N} \to \mathbb{N} \ !!!$$

That is to say, our equation picks *all* of the functions, not one. This is fine if you are specifically picking a class of functions and you don't care which one it is: logicians call this an *indefinite description*. But you'd better know that you're not naming a single function! This is not a function definition because it picks too many functions.

For a second broken example, suppose our equation is $F(n) = 1 + F(n)$. This one is broken for the opposite reason:

$$\{\, F \in \mathbb{N} \to \mathbb{N} \mid \forall n \in N.F(n) = 1 + F(n) \,\} = \emptyset!$$

There are *no* functions with this property. So this is not a definition because it picks too few functions. Definitions like this are particularly bad because we can prove all sorts of terribly wrong things by performing deductions using the description of a function that *doesn't exist*! Here's a fun example of what can go wrong:

$$\begin{aligned} 1 &= 1 && \text{by identity;} \\ F(1) &= F(1) && \text{by applying a function to two equals;} \\ F(1) &= 1 + F(1) && \text{by the ``definition'' of F;} \\ F(1) - F(1) &= 1 + F(1) - F(1) && \text{by subtracting equals from equals;} \\ 0 &= 1 && \text{by definition of minus.} \end{aligned}$$

So by reasoning with a function that we could never have, we prove something that's totally, albeit obviously, false. In this case we can see that something went terribly wrong, but what's really bad is when you "prove" something that isn't obviously false, but is false nevertheless. Granted the example above is harebrained, but plenty of prospective theorists have been led astray by morally doing exactly this kind of thing, and then thinking that they have proven an interesting theorem when in fact they have done no such thing because they started with a bad definition: assuming the existence of a mathematical object that does not exist.

The point is that when you try to define a function (or other single elements) by providing a set of properties, you are obliged to show that those properties *uniquely* characterize the function. The Principle of Recursion is a great workhorse because it once-and-for-all dispatches that obligation for those sets of equations that can be stated in the form that it discusses: as long as our equations fit the Principle of Recursion, we know that we have a real function definition. We call this a particular *recursion scheme*. We'll find that we can identify and prove additional recursion schemes, that let us identify new classes of functions that cannot be shoehorned into this particular scheme.

[17] Sadly I see it in research papers (and textbooks) all too often!

Now, whether the function you have successfully defined is the one that you really wanted is a different, more philosophically interesting question. For example, how would you go about arguing that the *trues* function *really does* count the number of trues in a term? That one isn't too bad, but in general, arguing that your formalization of a previously vague and squishy concept is "the right definition" is a matter of analytic philosophy, a rather challenging field of inquiry, but a common issue in the theory of programming languages and other areas of computer science.[18]

5.5 Boolean Language Evaluator, Revisited

One particular function over Boolean expressions is an *evaluator* for it, much like the evaluator function for the Vapid languages.. Conveniently enough, we can define the evaluator for this particular language as a recursive function:

$$\begin{gathered}
\textsc{Pgm} = \textsc{Term}, \quad \textsc{Obs} = \textsc{Value} \\
eval : \textsc{Pgm} \to \textsc{Obs}
\end{gathered}$$

$$\begin{aligned}
eval(\mathsf{true}) &= \mathsf{true} \\
eval(\mathsf{false}) &= \mathsf{false} \\
eval(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= eval(t_2)\ \text{if}\ eval(t_1) = \mathsf{true} \\
eval(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= eval(t_3)\ \text{if}\ eval(t_1) = \mathsf{false}
\end{aligned}$$

Recall that this evaluator definition is justified by the Principle of Definition by Recursion on Elements $t \in \textsc{Term}$, which means that we chose:

1. $S = \textsc{Value}$;
2. $s_t = \mathsf{true}$;
3. $s_f = \mathsf{false}$;
4. $H_{if} : \textsc{Value} \times \textsc{Value} \times \textsc{Value} \to \textsc{Value}$
   $H_{if}(\mathsf{true}, v_1, v_2) = v_1$
   $H_{if}(\mathsf{false}, v_1, v_2) = v_2$.

and fed them into the principle to produce a unique function, and we then convince ourselves intuitively that this is in fact our intended evaluator.

5.6 A Small Case Study

For more experience with inductive proof, let's revisit our *bools* function from earlier, which I simply claimed was properly defined. First let's see the *shorthand* style again:

Definition 6. *Let bools* : $\textsc{Term} \to \mathbb{N}$ *be defined by*

$$\begin{aligned}
bools(\mathsf{true}) &= 1 \\
bools(\mathsf{false}) &= 1 \\
bools(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= bools(t_1) + bools(t_2) + bools(t_3)
\end{aligned}$$

Based on the material up above, we can unpack this definition into the low-level pieces that correspond to the principle of recursion. Here is the *longhand* presentation:

1. $S = \mathbb{N}$;
2. $s_t = 1$, $s_f = 1$

[18] For example, what does it mean for a programming language to be *secure*? Philosophical debates on this issue abound!

3. $H_{if} : \mathbb{N} \times \mathbb{N} \times \mathbb{N}; H_{if}(n_1, n_2, n_3) = n_1 + n_2 + n_3$.

You should convince yourself that this function yields the number of falses and trues in a Term. At this point, the only way that you can do that is to use the function to reason about enough examples that you have confidence that your function meets your "informal specification." This is pretty much like programming practice: you need enough "unit tests" to convince yourself and others that you have the right specification. All we had done so far is prove that what we have in our hands is a *proper specification* of *some specific function*: whether it gives you what you want or not is a totally separate question!

A mathematician's "proof" We now have *two* principles for proving hard facts about Terms, and we have a principle for defining functions on Terms. To wrap things up, let's revisit the proof that $bools(t) > 0$. Here is the way that a mathematician would write that proof, at least using our shiny new principle of induction on elements $t \in$ Term. For reference, here is a statement and a proof of that statement as you would likely see it in a textbook:

Proposition 28. *$bools(t) > 0$ for all $t \in$ Term*

Proof. By induction on t

Case 17 (true). $bools(\text{true}) = 1 > 0$

Case 18 (false). Analogous to true.

Case 19 (if). If *bools* yields a positive number for each subcomponent of if then their sum will be positive too.

□

Wow, so shiny and tiny! Remember: this is pseudocode, not code: it's a proof sketch, not a precise formal proof.[19] Seeing the relation between the above conversational statements and the precise pedantic formal principal of induction that we presented above may not be all that obvious at first, but if you start from the principle above, you should be able to figure out a formal property P and recast each of the cases as one of the pieces of the statement of the principle of induction. The form you see here is typical of what shows up in the literature. It's important to be able to make that connection if you hope to really understand proofs and be able to check whether they are correct. Going forward, you will see more examples.

To better understand the connection, I recommend that you rewrite the above proof in the more precise (longhand) style to ensure that you can. To me this is akin to "I recommend that you implement the algorithms in your algorithms textbook to ensure that you can."

5.7 Parting Thoughts

To wrap up, the last couple of classes we have been addressing two issues. We have been introducing some preliminary notions from the semantics of programming languages, and at the same time establishing a common understanding for how the math underlying those semantics "works."

On the semantics front, we've talked about the idea of a language being defined as some set of programs (the "syntax" if you will), and a mapping from programs to observable results (the "semantics"). Our examples have been simple so far, but we've observed that whatever approach we use to define the evaluator has a significant impact on how we reason about our language and its programs.

On the mathematical front, we discussed some of the basic ways of building sets:

1. Extensionally, i.e., by enumerating elements, which works for a finite set (e.g., $\{1, 2\}$): Along with it comes reasoning by cases;

2. taking the union ($A \cup B$) or intersection ($A \cap B$) of sets that you already have (A and B): for these we reason by disjunction ("or") or conjunction ("and");

[19] I once heard a software engineering researcher tell the following joke about the Unified Modeling Language (UML), ascribing it to Bertrand Meyer: "Q: What's the good thing about bubbles and arrows, as opposed to programs? A: Bubbles and arrows never crash." I leave you to ponder the relevance.

3. forming the product $(A \times B)$ of two sets.
4. Using separation to filter the elements of some set $\{ a \in A \mid P(a) \}$ according to some predicate P. This too gives us a proof principle corresponding to the axiom of separation.

Inductive definitions with rules, and definition of functions by (recursive) equations, are simply particular instances of item (4) above, where the elements are filtered based on the existence of derivations and the satisfaction of those equations, respectively.

At this juncture we have enough mathematical machinery to draw our focus more on the programming language concepts. Any new mathematical concepts we need can be weaved in on demand.

Bibliography

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.

J. Avigad. Reliability of mathematical inference. *Synthese*, 2020. doi: 10.1007/s11229-019-02524-y. `https://doi.org/10.1007/s11229-019-02524-y`.

F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

J. Bagaria. Set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* The Metaphysics Research Lab, winter 2014 edition, 2014. URL `http://plato.stanford.edu/archives/win2014/entries/set-theory/`.

R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL `http://doi.acm.org/10.1145/3182657`.

H. P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics.* North-Holland, 1985. ISBN 978-0-444-86748-3.

A. Bauer. Proof of negation and proof by contradiction. Mathematics and Computation Blog, March 2010. `http://math.andrej.com/2010/03/29/proof-of-negation-and-proof-by-contradiction/`.

L. Crosilla. Set Theory: Constructive and Intuitionistic ZF. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Summer 2020 edition, 2020.

M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(89)90069-8. URL `http://www.sciencedirect.com/science/article/pii/0304397589900698`.

M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 1st edition, 2009.

J. Ferreirós. The early development of set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019. URL `https://plato.stanford.edu/archives/sum2019/entries/settheory-early/`.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563. URL `http://doi.acm.org/10.1145/512529.512563`.

P. R. Halmos. *Naive Set Theory.* Springer-Verlag, first edition, Jan. 1960. ISBN 0387900926.
A classic introductory textbook on set theory.

P. R. Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, New York, NY, USA, 2012. URL `http://www.cs.cmu.edu/%7Erwh/plbook/1sted-revised.pdf`.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124 (2):103–112, Feb. 1996. ISSN 0890-5401.

G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, UK, 1987. Springer-Verlag.

R. Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, December 2015. URL `https://robbertkrebbers.nl/thesis.html`.

C. Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1): 161–171, 1921. doi: 10.4064/fm-2-1-161-171. `https://web.archive.org/web/20190429103938/http://matwbn.icm.edu.pl/ksiazki/fm/fm2/fm2122.pdf`.

P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. doi: 10.1093/comjnl/6.4.308. URL `https://doi.org/10.1093/comjnl/6.4.308`.

X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, Feb. 2009. ISSN 0890-5401.

P. Maddy. Believing the axioms. I. *The Journal of Symbolic Logic*, 53(02):481–511, 1988.
An interesting (though complicated) analysis of why set theorists believe in their axioms.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

J. H. J. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, Feb. 1969. URL `http://hdl.handle.net/1721.1/64850`.

A. M. Pitts. *Nominal sets: Names and symmetry in computer science.* Cambridge University Press, 2013.

G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004a. doi: 10.1016/j.jlap.2004.03.009. URL `http://dx.doi.org/10.1016/j.jlap.2004.03.009`.

G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004b.

B. Popik. ”pull yourself up by your bootstraps”. Weblog entry, September 2012. `https://www.barrypopik.com/index.php/new_york_city/entry/pull_yourself_up_by_your_bootstraps/`.

E. Reck and G. Schiemer. Structuralism in the Philosophy of Mathematics. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Spring 2020 edition, 2020.

D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4): 15:1–15:41, May 2009. ISSN 0164-0925.

W. Sieg and D. Schlimm. Dedekind's analysis of number: Systems and axioms. *Synthese*, 147(1):121–170, Oct 2005.

J. Spolsky. The law of leaky abstractions. Joel on Software Blog, November 2002. `https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/`.

G. L. Steele. Debunking the “expensive procedure call”” myth or, procedure call implementations considered harmful or, lamdba: The ultimate goto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977. URL `http://dspace.mit.edu/handle/1721.1/5753`.

S. Stenlund. Descriptions in intuitionistic logic. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundations of Mathematics*, pages 197 – 212. Elsevier, 1975. URL `http://www.sciencedirect.com/science/article/pii/S0049237X08707328`.

M. Tiles. Book Review: Stephen Pollard. Philosophical Introduction to Set Theory. *Notre Dame Journal of Formal Logic*, 32(1):161–166, 1990.
A brief introduction to the philosophical issues underlying set theory as a foundation for mathematics.

C. Urban and M. Norrish. A formal treatment of the Barendregt variable convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*, MERLIN '05, page 25–32, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930728. doi: 10.1145/1088454.1088458. URL `https://doi.org/10.1145/1088454.1088458`.

C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, CADE-21, page 35–50, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540735946. doi: 10.1007/978-3-540-73595-3_4. URL `https://doi.org/10.1007/978-3-540-73595-3_4`.

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

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994.

B. Zimmer. figurative "bootstraps". email to linguistlist mailing list, August 2005. `http://listserv.linguistlist.org/pipermail/ads-l/2005-August/052756.html`.