Reasoning about Inductive Definitions: Inversion, Induction, and Recursion

CPSC 509: Programming Language Principles

Ronald Garcia∗

13 January 2014

(Time Stamp: 16:00, Friday 14th October, 2016)

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 how can we prove properties of all programs in the language and the results of evaluating them? To answer this question, we introduce three new reasoning principles: inversion lemmas, proofs by induction, and definitions of functions by recursion.

1 Derivation and Inversion

Recall the definition of the language of Boolean Expressions \( t \in \text{TERM} \subseteq \text{TREE} \):

\[
\text{true} \in \text{TERM} \quad \text{(r-true)} \\
\text{false} \in \text{TERM} \quad \text{(r-false)} \\
r_1 \in \text{TERM} \quad r_2 \in \text{TERM} \quad r_3 \in \text{TERM} \quad \text{if \ r}_1 \text{ then \ r}_2 \text{ else \ r}_3 \in \text{TERM} \quad \text{(r-if)}
\]

From the above, we know that \( \text{TERM} = \{ r \in \text{TREE} \mid \exists D. \ D :: r \in \text{TERM} \} \). (remember that the \( \in \text{TERM} \) on the right side of the comprehension is syntactic sugar, so this is not circular). So we know that for each element \( t \in \text{TERM} \), it is also true that \( t \in \text{TREE} \); furthermore, there must be at least one derivation \( D :: r \in \text{TERM} \). More formally, our definition of the set \( \text{TERM} \) gives us the reasoning principle:

\[
\forall t. t \in \text{TERM} \iff t \in \text{TREE} \land \exists D \in \text{DERIV}.D :: t \in \text{TERM}.
\]

We take advantage of this connection between derivations and \( \text{TERMS} \) to reason about the \( \text{TERMS} \) by proving things about derivations. First, we take the following property of our derivations as given.

**Proposition 1** (Principle of Cases on Derivations \( D :: r \in \text{TERM} \)).

If \( D :: r \in \text{TERM} \), then exactly one of the following is true:

1. \( D = \text{true} \in \text{TERM} \text{ (r-true)} \)

¹ In general, these language properties are interesting, but because Vapid is so...vapid, the properties are trivial.
2. $D = \text{false} \in \text{TERM}$ (r-false)

3. $D = \begin{array}{ccc}
D_1 & D_2 & D_3 \\
\text{if } r_1 \text{ then } & \text{else } & \text{for some derivations } D_1, D_2, D_3.
\end{array}$ (r-if)

Based on our understanding of inductive rules and derivations that use instances of these rules, the above statement is obviously true. We state it explicitly for two reasons. First, though we choose to take this rule as given directly from our inductive definition, this is not strictly necessary. I’ve mentioned that we can build the idea of inductive rules and derivations directly in set theory, where an inductive rule is some kind of set, and a definition is another kind of set, etc. If we were to do this, then the above principle would be a theorem of set theory, not a fact that we take as given. However, diving that deep is too much like programming directly in machine language: the bits that your CPU understands. At the least we can use something more akin to assembly 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, just assume that whenever you have an inductive definition, you get a principle of cases on the structure of derivations.

1.1 Inversion Lemmas

So now that we have this principle of cases, what can we do with it? Well, we can prove a set of lemmas about TERMS, that you will expect to be obviously true, but might not have been sure how to prove.

**Lemma 1** (Inversion on $r \in \text{TERM}$). If $r \in \text{TERM}$ then one of the following is true:

1. $r = \text{true}$
2. $r = \text{false}$
3. $r = \text{if } r_1 \text{ then } r_2 \text{ else } r_3$ for some $r_1, r_2, r_3 \in \text{TERM}$.

**Proof.** Since this is one of the first proofs you’ll see, I’m going to walk through it in painful detail, then rewrite it as you would typically see in a paper. The first is to help you understand the strategy, and the second is to help your writing.

Notice that we are proving an implication: if $r \in \text{TERM}$, then case 1 holds or case 2 holds or case 3 holds. To prove an implication, we assume the premise and try to prove the conclusion, so:

“Suppose some $r \in \text{TERM}$.”

Now we need to prove the consequence: “case 1 holds or case 2 holds or case 3 holds.”

To make progress, we observe that $r \in \text{TERM}$ can only be true if there is some $D :: r \in \text{TERM}$. So we have deduced the existence of $D$. Here we could say “Then there is a derivation $D$”, but typically we don’t bother in writing out these proofs. Most paper proofs skip over this observation because “by goodness WE KNOW that you’re using inductive definitions so WE KNOW there’s a derivation: get on with it!” But it’s useful to recognize, even if you don’t write it down, that you are taking a logical step here.

Next, now that we have a derivation $D$, (since we’ve concluded that one exists!) we apply the Principle of Cases on Derivations to our derivation to conclude that one of the following is true.

1. $D = \text{true} \in \text{TERM}$ (r-true)
2. $D = \text{false} \in \text{TERM}$ (r-false)
3. $D = \begin{array}{ccc}
D_1 & D_2 & D_3 \\
\text{if } r_1 \text{ then } & \text{else } & \text{for some derivations } D_1, D_2, D_3.
\end{array}$ (r-if)
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 hand, 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 disjunction is to say something like:

“We proceed by cases on the structure of \( D \)

And then write out the cases separately as follows.

Case. Suppose \( D = \text{true} \in \text{TERM} \) (r-true). Then clearly \( D : \text{true} \in \text{TERM} \), so \( r = \text{true} \). Thus we have proven that one of the three conclusions holds.

Case. Suppose \( D = \text{false} \in \text{TERM} \) (r-false). Then clearly \( D : \text{false} \in \text{TERM} \), so \( r = \text{false} \). Thus we have proven that one of the three conclusions holds.

Case. Suppose \( D = \frac{D_1}{r_1 \in \text{TERM}} \frac{D_2}{r_2 \in \text{TERM}} \frac{D_3}{r_3 \in \text{TERM}} \) (r-if) for some derivations \( D_1, D_2, D_3 \). Well clearly \( r = \text{if } r_1 \text{ then } r_2 \text{ else } r_3 \) \( \in \text{TERM} \), and from the three subderivations we deduce that \( r_1, r_2, r_3 \) \( \in \text{TERM} \).

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.

First, let me rewrite this proof as I would normally write it down. This mostly involves leaving out details that a seasoned theorem-prover will be able to fill in herself.

**Proof.** Suppose \( r \in \text{TERM} \). We then proceed by cases on the structure of \( D \)

Case \( (D = \text{true} \in \text{TERM} \) (r-true). Then \( r = \text{true} \) immediately.

Case \( (D = \text{false} \in \text{TERM} \) (r-false). Analogous to the previous case.

Case \( (D = \frac{D_1}{r_1 \in \text{TERM}} \frac{D_2}{r_2 \in \text{TERM}} \frac{D_3}{r_3 \in \text{TERM}} \) (r-if) \). Then \( r = \text{if } r_1 \text{ then } r_2 \text{ else } r_3 \) immediately, and from the three subderivations we deduce that \( r_1, r_2, r_3 \) \( \in \text{TERM} \).

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, \( \{ \top \} \vdash \top \text{ true} \). 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. Usually we won’t bother proving these inversion lemmas, because the proof is always done the same way, and you expect it to be true. Nonetheless, these lemmas are used a lot going forward so you should know how to prove them.

Second, these inversion lemmas are sometimes useful when thinking about implementing artifacts related to the proof rules. In the case of \( r \in \text{TERM} \), what we basically get is a strategy for *implementing a parser* for Terms. Essentially, for our purposes, a parser is a program that given some tree \( r \), tries to build a derivation \( D : r \in \text{TERM} \) starting from the bottom and working upwards. If you 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 for evaluating programs or for specifying which programs are well-typed, we will specialize the inversion lemmas to assume certain inputs (e.g. input program) and yield certain outputs (e.g., the result of evaluation).
2 Inductive Reasoning

The strategy that we use to define an evaluator 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 members of the set of terms. For our purposes, an inductive definition automatically gives us reasoning principles tailored to the particular definition. The reasoning principle for Boolean Expressions follows.

Proposition 2 (Principle of Structural Induction on Derivations \(D :: t \in \text{TERM}\)).
Let \(P\) be a predicate on derivations \(D\). Then \(P(D)\) holds for all derivations \(D\) if:

1. \(P\left(\text{true} \in \text{TERM} \quad (r\text{-true})\right)\) holds;
2. \(P\left(\text{false} \in \text{TERM} \quad (r\text{-false})\right)\) holds;
3. If \(P\left(D_1 \cdot r_1 \in \text{TERM}\right), P\left(D_2 \cdot r_2 \in \text{TERM}\right),\) and \(P\left(D_3 \cdot r_3 \in \text{TERM}\right)\) hold then

\[
P\left(\frac{D_1 \cdot r_1 \in \text{TERM} \quad D_2 \cdot r_2 \in \text{TERM} \quad D_3 \cdot r_3 \in \text{TERM}}{(\text{r-if})}\right)\]

holds.

Proof. For our purposes, this comes for free with the inductive definition of \(t \in \text{TERM}\).

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 \(P\) 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 holds for the 3 subderivations of the \(\text{if}\) derivation and then holds for the whole derivation itself. The first two cases are a little different. Since the \((r\text{-true})\) and \((r\text{-false})\) rules have no premises, it’s vacuously true that the property holds for all of the subderivations: because there are none.

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 \(P(D)\) holds for any particular \(D\): If we know that \(P\) holds for any derivation that is exactly an axiom, and we know that whenever we combine derivations \(D_i\) 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 leaves 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 \(P(D)\) holds.

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, but 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, let’s immediately use the principle of induction on derivations to establish that we can safely reason about terms directly.

Proposition 3 (Principle of Structural Induction on Elements \(t \in \text{TERM}\)).
Let \(P\) be a predicate on terms \(t\). Then \(P(t)\) holds for all terms \(t\) if:

1. \(P(\text{true})\) holds;
2. \(P(\text{false})\) holds;
3. If \( P(t_1) \) and \( P(t_2) \) hold then \( P(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) \) holds.

Proof. Suppose we have a property \( P \) as described above. Then define a property \( Q \) of derivations as follows:

\[
Q(D) \equiv P(t) \quad \text{where } D :: t \in \text{TERM}.
\]

We can prove by Proposition 2 that \( Q(D) \) holds for all derivations \( D \):

1. \( Q \left( \text{true} \in \text{TERM} \right) \) holds since \( P(\text{true}) \) holds;

2. \( Q \left( \text{false} \in \text{TERM} \right) \) holds since \( P(\text{false}) \) holds;

3. Suppose that \( Q \left( r_1 \in \text{TERM} \right) \), \( Q \left( r_2 \in \text{TERM} \right) \), and \( Q \left( r_3 \in \text{TERM} \right) \) hold for three derivations \( D_1, D_2, D_3 \).

Then by the definition of \( Q(D) \) we know that \( P(r_1), P(r_2), \) and \( P(r_3) \) must hold. By our assumptions about \( P \), it follows that \( P(\text{if } r_1 \text{ then } r_2 \text{ else } r_3) \) holds. Then, appealing to our definition of \( Q \), we conclude that

\[
Q \left( \frac{D_1 \quad D_2 \quad D_3}{\begin{array}{c} r_1 \in \text{TERM} \quad r_2 \in \text{TERM} \quad r_3 \in \text{TERM} \\ \text{if } r_1 \text{ then } r_2 \text{ else } r_3 \in \text{TERM} \end{array}} \right)
\]

holds.

Now appealing to the Principle of Structural Induction on Derivations, \( Q \) holds for all derivations \( D \). By definition of \( Q \), this means that \( P(t) \) holds for every term \( t \) with a derivation \( D \), but remember that the set of terms is defined by the characteristic of having a derivation. Thus, \( P(t) \) holds for all terms \( t \). \( \square \)

Proposition 3 uses the principle of induction on derivations to establish another theorem with the same general structure that lets us establish a property of all \( \text{TERM} \)s by proving a few small things. We’ll see later in the course that there are many kinds of principles of induction, all of which fit a more general characterization, but for now these two kinds should get us pretty far.

A final note. Consider again Proposition 1, the principle of cases on derivations. We can prove it by induction on derivations \( D \), and in doing so, we never make use of any of the induction hypotheses. In that sense, proof by cases on derivations is a degenerate variant of proof by induction on derivations.

### 2.1 Induction on Propositional Entailment

To give you another example of induction on elements, as we are calling it, here is the statement of the Principle of Induction on Elements \( \Gamma \vdash p \text{ true} \) (i.e. elements \( (\Gamma, p) \in \vdash \cdot \text{ true} \)). It is proven true in terms of the Principle of Induction on Derivations of \( \mathcal{D} :: \Gamma \vdash p \text{ true} \), which we leave as an exercise for you to state. To keep it manageable, we will focus on just the fragment of propositional logic \( p \in \text{PROP} \) corresponding to \( p ::= A \mid \bot \mid p \land p \mid p \lor p \).

**Proposition 4** (Principle of Induction on Elements of Entailment). Let \( P \) be a property of entailments \( \Gamma \vdash p \text{ true} \). Then \( P \) holds for all \( \Gamma, p \) such that \( \Gamma \vdash p \text{ true} \) if

1. \( P(\Gamma \vdash p \text{ true}) \) holds whenever \( p \in \Gamma \).

2. If \( P(\Gamma \vdash \bot \text{ true}) \) holds then \( P(\Gamma \vdash p \text{ true}) \) holds for all \( p \in \text{PROP} \).

3. If \( P(\Gamma \vdash p_1 \text{ true}) \) and \( P(\Gamma \vdash p_2 \text{ true}) \) hold then \( P(\Gamma \vdash p_1 \land p_2 \text{ true}) \) holds.

4. If \( P(\Gamma \vdash p_1 \land p_2 \text{ true}) \) holds then \( P(\Gamma \vdash p_1 \text{ true}) \) holds and \( P(\Gamma \vdash p_2 \text{ true}) \) holds.

5. If \( P(\Gamma \vdash p_1 \lor p_2 \text{ true}) \) holds then \( P(\Gamma \vdash p_1 \supset p_2 \text{ true}) \) holds.

6. If \( P(\Gamma \vdash p_1 \supset p_2 \text{ true}) \) and \( P(\Gamma \vdash p_1 \text{ true}) \) hold then \( P(\Gamma \vdash p_2 \text{ true}) \) holds.
One side-note. Notice that Proposition 3 is said to be a principle of structural induction while Proposition 4 is not. That’s because Proposition 3 follows the structure of the abstract syntax exactly: we just have to show that the property holds for an expression if it holds for each immediate subexpression. Similarly, Proposition 2 follows the structure of a derivation. On the other hand, the cases for Proposition 4 do not uniformly follow the structure of propositions \( p \), sets of propositions \( \Gamma \), or even pairs. Nonetheless the principle is justifiable. Later in the course we will talk about how we can come up with new induction principles for old sets.

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.

3 Defining Functions

We’ve now developed a tool that we can use to prove properties of terms, but we have no experience using it yet, and furthermore we still need a way to define functions on terms. As it turns out, we can kill two birds with one stone: we will use Proposition 3 to establish a principle for defining functions over our infinite set of programs, and we’ll use the result to produce our first non-trivial function over an inductively-defined infinite set. We’ll also see that the structure of this function definition will allow use to reason about its properties, including calculate how this function maps particular inputs to outputs.

**Proposition 5** (Principle of Definition by Recursion on terms \( t \in \text{TERM} \)). Let \( S \) be a set and \( s_t, s_f \in S \) be two elements and

\[ H_{df} : S \times S \times S \to S \]

be a function on \( S \). Then there exists a unique function

\[ F : \text{TERM} \to S \]

such that

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

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 5 to define a function! According to the proposition all we need is:

1. some set \( (S) \);
2. two elements \( (s_t \text{ and } s_f) \) of that set, though you can use the same element for both; and
3. some function \( (H_{df}) \) 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 \( s_t \text{, and } 0 \text{ for } s_f \): \( s_t = 1 \) 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 3, there is a unique function \( F : \text{TERM} \to \mathbb{N} \) with the properties that:

\[
F(\text{true}) = 1; \quad (1)
\]
\[
F(\text{false}) = 0; \quad (2)
\]
\[
F(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) = H(F(t_1), F(t_2), F(t_3)) = F(t_1) + F(t_2) + F(t_3). \quad (3)
\]
That means that these three equations (properties) uniquely characterize some function from TERMS to functions. At this point all we know is that there is a function that satisfies these properties, and that there’s only one. It’s a black box to us except for these equations, which tell us something about the function. 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. Instead, I hope you’ll trust me that this function maps every TERM in our language to the number of trues in it. We can 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 part to 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.²

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

**Definition 1.** The function trues : TERM → N is defined by

\[
\text{trues}(\text{true}) = 1 \\
\text{trues}(\text{false}) = 0 \\
\text{trues}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) = \text{trues}(t_1) + \text{trues}(t_2) + \text{trues}(t_3).
\]

The function is given by the final specialized version of the equations, and it’s up to you (the reader) to figure out which \(S, s_i, s_f\) and \(H_f\) 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:

\[
\text{trues} \in \{ F \in \text{TERM} \to \text{N} \mid P(F) \}
\]

Where \(P(F)\) is the combination of equations (1)-(3) above. The important thing is that in order for \(P(F)\) to be a good definition, 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!³ Let’s consider two simple examples. Take the natural numbers \(\text{N}\), and suppose I claim I’m defining a function \(F : \text{N} \to \text{N}\) by the equation \(F(n) = F(n)\). Well that doesn’t define a unique function at all because

\[
\{ F \in \text{N} \to \text{N} \mid \forall n \in \text{N}.F(n) = F(n) \} = \text{N} \to \text{N} !!!
\]

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, 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:

\[
\{ F \in \text{N} \to \text{N} \mid \forall n \in \text{N}.F(n) = 1 + F(n) \} = \emptyset!
\]

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 taking reasoning steps that use a function that doesn’t exist! Here’s a fun example of what can go wrong:

\[
\begin{align*}
1 &= 1 & \text{by identity;} \\
F(1) &= F(1) & \text{by applying a function to two equals;} \\
F(1) &= 1 + F(1) & \text{by the “definition” of } F; \\
F(1) - F(1) &= 1 + F(1) - F(1) & \text{by subtracting equals from equals;} \\
0 &= 1 & \text{by simple arithmetic.}
\end{align*}
\]

²This shorthand and longhand terminology is my own creation. I don’t think you’ll find it in the literature.
³Sadly I see it in research papers (and textbooks) all too often!
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 hairbrained, 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.

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. 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 \textit{trues} function really does count the number of \textit{true}s 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.

4 A Small Case Study

For more experience, let’s define a recursive function, first using the shorthand style:

\begin{definition}
Let \( \text{bools} : \text{TERM} \to \mathbb{N} \) be defined by

\[
\begin{align*}
\text{bools}(\text{true}) &= 1 \\
\text{bools}(\text{false}) &= 1 \\
\text{bools}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{bools}(t_1) + \text{bools}(t_2) + \text{bools}(t_3)
\end{align*}
\end{definition}

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. \( s_t = 1, s_f = 1 \)
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 \textit{false}s and \textit{true}s in a \text{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.

Armed with this definition, you can consider size to be a property of \text{TERM}s, so we can actually prove something about that property, that every \text{TERM}’s size is greater than 0. Here is a statement and a proof of that statement as you would likely see it in a textbook

\begin{proposition}
\( \text{bools}(t) > 0 \) for all \( t \in \text{TERM} \)
\end{proposition}

\begin{proof}
By induction on \( t \)
\begin{enumerate}
\item \textbf{Case (true).} \( \text{bools}(\text{true}) = 1 > 0 \)
\item \textbf{Case (false).} Analogous to \textit{true}.
\item \textbf{Case (if).} If \( \text{bools} \) yields a positive number for each subcomponent of \textit{if} then their sum will be positive too.
\end{enumerate}
\end{proof}

Seeing the relation between the above conversational statements and the precise formal principal of induction that we presented last time 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.

5 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. enumerating elements, which works for a finite set (e.g., \{1, 2\});
2. taking the union (A ∪ B) or intersection (A ∩ B) of sets that you already have (A and B);
3. forming the product (A × B) of two sets.
4. filtering the elements of some set \{a ∈ A | P(a)\} according to some predicate P.

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.

References
