CSE396 Structural Induction on CFGs Spring 2018

A context-free grammar G is *sound* for a language T if $L(G) \subseteq T$. The original meaning of *sound* in logic is actually very similar: in place of G we have more-general notion of a *formal system* F that uses rules of logical proof to generate *theorems* and T is the set of statements that are true. So F is sound if everything it proves is really true. With grammars the meaning is that everything it derives really belongs to the language.

Put another way, soundness means the grammar avoids "false positives." In many situations this is more important than G being comprehensive, which would mean $T \subseteq L(G)$. Comprehensiveness is also harder to prove—indeed, we will see that related problems are *undecidable* in general. And in the wider realm of logic there's a flat "no" answer: no F with verifiable proof rules can be both sound and comprehensive for mathematical truth. The word used for the latter in logic is "complete," which in our course would be confusing so I prefer saying "comprehensive." But if you use the logic word in the negative then you can restate this as saying that any F that is sound and has verifiable proof rules must be "incomplete"—there must be true statements that it can formulate but it *cannot prove*. In case you've wondered what *Gödel's Incompleteness Theorem* is, that's pretty much it. We will not cover it *per se*, but it inspired Alan Turing to prove *undecidability* which is the upshot of chapter 5 of the text.

Back to grammars and soundness. There is a proof technique for showing soundness called *Structural Induction* (SI) which has various forms but is (IMHO) most naturally associated with grammars. The point is to think inductively directly on the rules of the grammar rather than on "n going to $n+1$" in the natural numbers. This kind of thinking extends readily to rules that arise in object-oriented design, where making a subclass is a kind of "derivation" from the parent class. You want to verify that the subclass objects obey the properties and behaviors mandated by the parent class and any other classes or interfaces that got "mixed in." You usually don't need your subclass to be "comprehensive" in any way—what you need is the new objects to be *sound*. The goal is a logically organic feel to your class hierarchy. Well, here we are taking an organically logical approach to the grammars.

You actually have already seen a long proof by SI in the course already. It was by induction on the following context-free grammar G_{reg}:

$$E \to \emptyset \mid \epsilon \mid 0 \mid 1 \mid (E \cup E) \mid (E \cdot E) \mid (E^*).$$

Here $\emptyset, \epsilon, 0, 1$ as well as the operators $\cup, \cdot, *$ are being treated as *terminal symbols* along with the parentheses. You may remember I originally wrote the first four with squiggles underneath to say that they were special regular-expression characters. Now let T denote the set of regular expressions r such that there is an NFA N giving $L(N) = L(r)$. What the lecture proved was $L(G_{reg}) \subseteq T$, namely, every regular expression has a corresponding NFA. There is a 'vice-versa' insofar as every NFA has an equivalent regular expression, but that came later. Just for the proof in which I made an electrical-circuit analogy, the point is that it (a) used induction but (b) *never* mentioned a natural number "n." What it inducted on was how regular expressions are built up as formulas—which are precisely the rules of the grammar G_{reg}.

When the grammar has more than one variable, then what we are technically doing is a multi-threaded induction. In numerical terms like in CSE191 that could seem wickedly complicated. But when done on grammars, here is my "scripted" way to make it come naturally. I actually encourage "personifying" each variable as if it is some kind of agent. Here is the "proof script" in general given a grammar $G = (V, \Sigma, \mathcal{R}, S)$ and target language T:

1. Assign to each variable A a property P_A of strings it can derive. You can think of P_A as the "meaning" you give to A—though it need not be a comprehensive meaning, just enough to work with the other variables. You can personify it in the form, "P_A: Every x I derive is such that..."

2. The meaning P_S of the start symbol should imply $x \in T$. Often you can simply take P_S to be the assertion, "Every x I derive belongs to T," but sometimes you need to use a sharper property. Technically this is called "strengthing" or "loading" the "induction hypothesis" (IH), but what I call "SI style" tries to make it more transparent.

3. For each variable A and each rule $A \to X$ where $X \in (\Sigma \cup V)^*$, begin the body of the script by writing, "Suppose $A \Longrightarrow^* x$ using this rule first" (you can abbreviate the last four words to "utrf" or to "utpf" for "using this production first").

4. If X is already an all-terminal string x, you immediately need to check that x obeys what P_A says. This is a base case.

5. Otherwise, X has one or more *occurrences* of variables. Note that the variable A itself can occur on the right-hand side of the rule. This may seem like circular logic but it's not. It or some other variable(s) can occur more than once. Regardless, for each occurrence—call it B_i—let u_i (or any letter you like) stand for a corresponding substring that it derives. You have to be general and you have to include any terminals in the rule in the right order. If there are k occurrences of variables on the right-hand side (RHS) of the rule $A \to X$, then the script says to enuncuiate it by saying:

   "Then $x =: \dots u_1 \dots u_i \dots u_k \dots$ where $B_1 \Longrightarrow^* u_1$ and ... and $B_i \Longrightarrow^* u_i$ and ... $B_k \Longrightarrow u_k$."

6. Now we apply the corresponding properties $P_1, \dots, P_i, \dots, P_k$ of the variables on the RHS. Again, two of those occurrences may be the same variable, so you will use the same property twice, but you will generally be using the property *on* different substrings. Here is the scripted way to enunciate this:

   "By IH P_1 on RHS, the substring u_1 satisfies ..., and by IH P_2 on RHS, u_2 satisfies ..." And so on through all the substrings.

7. Finally, you need to *argue* that the fact of each substring obeying its property *ensures* that the resulting string x (whatever it is—it is general) obeys the original property P_A of the variable A on the left-hand side (LHS) of the rule. You can then summarize by saying, "This upholds P_A on LHS."

8. When you show that **every** rule upholds the stated property of its left-hand variable, then you uphold P_S in particular, which is what entitles you to conclude "$L(G) \subseteq T$ by structural induction."

You can't, however, just uphold the rules for S, because your reasoning might lean on other variables upholding their own properties. Well, when there are no other variables then not to worry—here's a simple example:

$$G = S \to (S)S \mid (), \quad T = \{\text{balanced-parentheses strings}\}.$$

This grammar is not comprehensive, but that's not what we're worrying about. Take $P_S =$ "Every x I derive is balanced." Now go through the rules:

- $S \to ()$: The terminal string () is balanced, so P_S on LHS is immediately upheld.
- $S \to (S)S$: Suppose $S \Longrightarrow^* x$ using this rule first. Then $x =: (u)v$ where $S \Longrightarrow^* u$ and $S \Longrightarrow^* v$. By IH P_S on RHS *twice*, u is balanced and v is balanced. It follows that x is balanced, **because** putting parens around the balanced string u is still balanced and concatenating that to the balanced string v is still balanced. This upholds P_S on LHS.

Since we upheld the meaning of each rule, every string in $L(G)$ is balanced, i.e., $L(G) \subseteq T$. $\Box$

Now we *could* formulate this as an induction on natural numbers n. Define the numerical predicate $P(n)$ to state that every x derivable in n steps is balanced. The basis is $P(1)$ since () is the only terminal string derivable in one step. Now let $n > 1$. We had to start with the rule $S \to (S)S$. Thus x breaks down as $(u)v$ where $S \Longrightarrow^* u$ in some number m_1 of steps and $S \Longrightarrow^* v$ in some number m_2 of steps. Adding up steps and remembering that the initial rule counts as one step, we get $1 + m_1 + m_2 = n$. This in particular means that both m_1 and m_2 are strictly less than n. Hence we can apply what is often called "strong induction" or "course-of-values induction" to allow taking $P(m_1)$ and $P(m_2)$ as valid induction hypotheses. These hypotheses say that u and v are balanced. We then flow into the same reasoning as above, except that what we conclude is "$P(n)$." We finally conclude that "$(\forall n)P(n)$ follows by induction," which tells us what we needed to prove.

What SI does is remove the clunkiness of referring explicitly to "n." It enunciates only the logic about how u and v combine with two parens to give a balanced x, which the latter "CSE191-style" proof needed anyway. By homing in on that logic, it helps us see that it holds also if we change the terminal rule from $S \to ()$ to $S \to \epsilon$: since ϵ counts as balanced, the grammar remains sound. This helps us visualize what liberating changes are OK (this particular change happens to make makes this grammar become comprehensive). The savings and ability to "brainstorm" changes are even better exemplified by a bigger example:

Let G be the grammar with variables A, B, S, terminals $0, 1$, and rules

$$\begin{array}{rcl} S & \to & SS \mid 0B \mid 1A \mid \epsilon \\ A & \to & 0S \mid 1AA \\ B & \to & 1S \mid 0BB \end{array}$$

Let $T = \{x \in \{0,1\}^* : \#0(x) = \#1(x)\}$. Prove that $L(G) \subseteq T$.

To do so, we have to first think up some appropriate properties. Well, we can bank on P_S stating "Every x I derive has equal 0s and 1s," but what about P_A and P_B? The rules $A \to 0S$ and $B \to 1S$, in order to be sound, suggest the following properties:

- P_A: Every y I derive has one more 0 than 1.
- P_B: Every y I derive has an extra 1 instead.

Now we can check the rules in any order we like—we don't have to consciously think of $S \to \epsilon$ as the basis, though it must be since it is the only terminal rule here.

- $S \to SS$: Suppose $S \Longrightarrow^* x$ using this rule first. Then $x =: uv$ where $S \Longrightarrow^* u$ and $S \Longrightarrow^* v$. By IH P_S on RHS (twice), u and v have equal 0s and 1s. Hence so does their concatenation x, which upholds P_S on LHS.
- $S \to 0B$: Suppose $S \Longrightarrow^* x$ utrf. Then $x = 0y$ where $B \Longrightarrow^* y$. By IH P_B on RHS, y has one more 1 than 0. The leading 0 in $0y$ brings the number of 0s in x up to equal the number of 1s in x (which is the same as in y), upholding P_S on LHS.
- $S \to 1A$: Suppose $S \Longrightarrow^* x$ utrf. Then $x = 1y$ where $A \Longrightarrow^* y$. By IH P_A on RHS, y has one more 0 than 1. The leading 1 in $1y$ brings the number of 1s in x up to equal the number of 0s in x (which is the same as in y), upholding P_S on LHS. (I mouse-copied the previous text and switched 0 with 1 and A with B.)
- $S \to \epsilon$: $\#0(\epsilon) = 0 = \#1(\epsilon)$, so P_S is immediately upheld.
- $A \to 0S$: Suppose $A \Longrightarrow^* y$ utrf. Then $y = 0z$ where $S \Longrightarrow^* z$. By IH P_S on RHS, z has equal 0s and 1s. So $0z$ supplies the extra 0 needed to uphold P_A on LHS.
- $A \to 1AA$: Suppose $A \Longrightarrow^* y$ utrf. Then $y = 1uv$ where $A \Longrightarrow^* u$ and $A \Longrightarrow^* v$. By IH P_A on RHS (twice!), u and v each supply an extra 0. Having two extra 0s offsets the leading 1 in the rule to give y a net of one extra 0, which upholds P_A on LHS in this case also.
- $B \to 1S$: Similar to the treatment of $A \to 0S$.
- $B \to 0BB$: Likewise with $A \to 1AA$. These are examples of "reasonable shortcuts."

Since we upheld every rule, $L(G) \subseteq T$ follows. $\square$

In this case, we happen to have comprehensiveness too, so actually $L(G) = T$. Proving in return that $T \subseteq L(G)$—i.e., that G catches *every* binary string with equal 0s and 1s—is a separate matter. It is also more painful: it requires *general parsing* of strings. Put another way, it requires designing a *parsing algorithm* dedicated to the grammar. The verification of the algorithm has to be by induction *directly on strings and substrings* that are being parsed. It can't be induction *on* derivations—working *from* derivations—because that would be like assuming what you are trying to prove. Sometimes this can be done "organically on strings" but there's less savings—you often might as well take "n" to refer to the *length* of a given string $x \in T$ and work in terms of the lengths $m_1, m_2, \dots$ of substrings you parse x into. The text doesn't really cover this (either)—it has a few end-of-chapter problems that presume you can do it based on background assumed from chapter 0. What it really has is the new section 2.4 with material on particular cases of parsing algorithms. We will not go into this or into comprehensiveness proofs in general. But these notes and lecture coverage should suffice as text for the *soundness* side.