Lecture Notes on Inductive Definitions

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 2
September 2, 2004

These supplementary notes review the notion of an inductive definition and give some examples of rule induction. References to Robert Harper's draft book on *Programming Languages: Theory and Practice* are given in square brackets, by chapter or section.

Given our general goal to define and reason about programming languages, we will have to deal with a variety of description tasks. The first is to describe the grammar of a language. The second is to describe its static semantics, usually via some typing rules. The third is to describe its dynamic semantics, often via transitions of an abstract machine. On the surface, these appear like very different formalisms (grammars, typing rules, abstract machines) but it turns out that they can all be viewed as special cases of *inductive definitions* [Ch. 1]. Following standard practice, inductive definitions will be presented via judgments and inference rules providing evidence for judgments.

The first observation is that context-free grammars can be rewritten in the form of inference rules [Ch. 3.2]. The basic judgment has the form

$$s\ A$$

where s is a string and A is a non-terminal. This should be read as the judgment that *s is a string of syntactic category A*.

As a simple example we consider the language of properly matched parentheses over the alphabet $\Sigma = \{\,\texttt{(},\texttt{)}\,\}$. This language can be defined by the grammar

$$M ::= \varepsilon \mid \texttt{(}M\texttt{)} \mid M\,M$$

with the only non-terminal M. Recall that ε stands for the empty string.

Rewritten as inference rules we have:

$$\frac{}{\varepsilon\ M}\,m_1 \qquad \frac{s\ M}{\texttt{(}s\texttt{)}\ M}\,m_2 \qquad \frac{s_1\ M \quad s_2\ M}{s_1\,s_2\ M}\,m_3$$

As an example, consider a deduction of `( ) ( )` M.

$$\dfrac{\dfrac{\dfrac{}{\varepsilon\ M}\,m_1}{\texttt{( )}\ M}\,m_2 \quad \dfrac{\dfrac{}{\varepsilon\ M}\,m_1}{\texttt{( )}\ M}\,m_2}{\texttt{( ) ( )}\ M}\,m_3$$

Our interpretation of these inference rules as an inductive definition of the judgment $s\ M$ for a string s means:

> *$s\ M$ holds if and only if there is a deduction of $s\ M$ using rules (m_1), (m_2), and (m_3).*

Based on this interpretation we can prove properties of strings in the syntactic category M by rule induction. To apply rule induction we have to show that the property in question is preserved by every inference rule of the judgment $s\ M$. That is, we have to show that for each rule, if all premises satisfy the property then the conclusion also satisfies the property. Here is a very simple example.

Theorem 1 (Counting Parentheses)
If $s\ M$ then s has the same number of left and right parentheses.

Proof: By rule induction. We consider each case in turn.

(Rule m_1) Then $s = \varepsilon$.

s has 0 left and 0 right parens	Since $s = \varepsilon$

(Rule m_2) Then $s =$ `(`s'`)`.

$s'\ M$	Subderivation
s' has n' left and right parens for some n'	By i.h.
s has $n' + 1$ left and right parens	Since $s =$ `(`s'`)`

(Rule m_3) Then $s = s_1\, s_2$.

$s_1\ M$	Subderivation
$s_2\ M$	Subderivation
s_1 has n_1 left and right parens for some n_1	By i.h.
s_2 has n_2 left and right parens for some n_2	By i.h.
s has $n_1 + n_2$ left and right parens	Since $s = s_1\, s_2$

■

The grammar we gave, unfortunately, is ambiguous [Ch. 3.3]. For example, there are infinitely many derivations that $\varepsilon\ M$, because

$$\varepsilon = \varepsilon\,\varepsilon = \varepsilon\,\varepsilon\,\varepsilon = \cdots$$

In the particular example of this grammar we would be able to avoid rewriting it if we can show that the abstract syntax tree [Ch. 4] we construct will be the same, independently of the derivation of a particular judgment.

An alternative is to rewrite the grammar so that it defines the same language of strings, but the derivation of any particular string is uniquely determined. The following grammar accomplishes this: [1]

$$L \quad ::= \quad \varepsilon \mid \texttt{(}\,L\,\texttt{)}\ L$$

One can think of L as a (possibly empty) list of parenthesized expressions, terminated by the empty string. This readily translates into an inductive definition via inference rules.

$$\frac{}{\varepsilon\ L}\ l_1 \qquad\qquad \frac{s_1\ L \qquad s_2\ L}{\texttt{(}\,s_1\,\texttt{)}\ s_2\ L}\ l_2$$

Now there are two important questions to ask: (1) is the new grammar really equivalent to the old one in the sense that it generates the same set of

[1] An alternative solution, suggested in lecture in 2003, exemplifies the idea of a simultaneous inductive definition. It uses two non-terminals L and N, where the category L corresponds to M, while N is an auxiliary non-terminal.

$$\begin{array}{rcl} L & ::= & \varepsilon \mid N\,L \\ N & ::= & \texttt{(}\,L\,\texttt{)} \end{array}$$

Note that the new definition arises from substituting out the definition of N in the alternation for L.

strings, and (2) is the new grammar really unambiguous. The latter is left as a (non-trivial!) exercise; the first one we discuss here.

At a high level we want to show that for any string s, $s\ M$ iff $s\ L$. We break this down into two lemmas. This is because "if-and-only-if" statements can rarely be proven by a single induction, but require different considerations for the two directions.

We first consider the direction where we assume $s\ M$ and try to show $s\ L$. When writing out the cases we notice we need an additional lemma. As is often the case, the presentation of the proof is therefore different from its order of discovery. To read this proof in a more natural order, skip ahead to Lemma 3 and pay particular attention to the last step in the case of rule (m_3). That step motivates the following lemma.

Lemma 2 (Concatenation)
If $s_1\ L$ and $s_2\ L$ then $s_1\, s_2\ L$.

Proof: By induction on the derivation of $s_1\ L$. Note that induction on the derivation on $s_2\ L$ will not work in this case!

(Rule l_1) Then $s_1 = \varepsilon$.

$s_2\ L$	Assumption
$s_1\, s_2\ L$	Since $s_1\, s_2 = \varepsilon\, s_2 = s_2$

(Rule l_2) Then $s_1 = \texttt{(}\,s_{11}\,\texttt{)}\ s_{12}$.

$s_{11}\ L$	Subderivation
$s_{12}\ L$	Subderivation
$s_2\ L$	Assumption
$s_{12}\, s_2\ L$	By i.h.
$\texttt{(}\,s_{11}\,\texttt{)}\ s_{12}\, s_2\ L$	By rule (l_2)

■

Now we are ready to prove the left-to-right implication.

Lemma 3
If $s\ M$ then $s\ L$.

Proof: By induction on the derivation of $s\ M$.

(Rule m_1) Then $s = \varepsilon$.

$s\ L$	By rule (l_1) since $s = \varepsilon$

(Rule m_2) Then $s = \texttt{(}s'\texttt{)}$.

$s'\ M$	Subderivation
$s'\ L$	By i.h.
$\varepsilon\ L$	By rule (l_1)
$\texttt{(}s'\texttt{)}\ L$	By rule (l_2) and $\texttt{(}s'\texttt{)}\ \varepsilon = \texttt{(}s'\texttt{)}$

(Rule m_3) Then $s = s_1\, s_2$.

$s_1\ M$	Subderivation
$s_2\ M$	Subderivation
$s_1\ L$	By i.h.
$s_2\ L$	By i.h.
$s_1\, s_2\ L$	By concatenation (Lemma 2)

∎

The right-to-left direction presents fewer problems.

Lemma 4
If $s\ L$ then $s\ M$.

Proof: By rule induction on the derivation of $s\ L$. There are two cases to consider.

(Rule l_1) Then $s = \varepsilon$.

$s\ M$	By rule (m_1), since $s = \varepsilon$

(Rule l_2) Then $s = \texttt{(}s_1\texttt{)}\ s_2$.

$s_1\ L$	Subderivation
$s_2\ L$	Subderivation
$s_1\ M$	By i.h.
$\texttt{(}s_1\texttt{)}\ M$	By rule (m_2)
$s_2\ M$	By i.h.
$\texttt{(}s_1\texttt{)}\ s_2\ M$	By rule (m_3)

■

Now we can combine the preceding lemmas into the theorem we were aiming for.

Theorem 5
$s\ M$ if and only if $s\ L$.

Proof: Immediate from Lemmas 3 and 4. ■

Some advice on inductive proofs. Most of the proofs that we will carry out in the class are by induction. This is simply due to the nature of the objects we study, which are generally defined inductively. Therefore, when presented with a conjecture that does not follow immediately from some lemmas, we first try to prove it by induction as given. This might involve a choice among several different given objects or derivations over which we may apply induction. If one of them works we are, of course, done. If not, we try to analyse the failure in order to decide if (a) we need to seperate out a *lemma* to be proven first, (b) we need to *generalize the induction hypothesis*, or (c) our conjecture might be false and we should look for a *counterexample*.

Finding a lemma is usually not too difficult, because it can be suggested by the gap in the proof attempt you find it impossible to fill. For example, in the proof of Lemma 3, case (Rule m_3), we obtain $s_1\ L$ and $s_2\ L$ by induction hypothesis and have to prove $s_1\, s_2\ L$. Since there are no inference rules that would allow such a step, but it seems true nonetheless, we prove it as Lemma 2.

Generalizing the induction hypothesis can be a very tricky balancing act. The problem is that in an inductive proof, the property we are trying to establish occurs twice: once as an inductive assumption and once as a conclusion we are trying to prove. If we strengthen the property, the induction hypothesis gives us more information, but conclusion becomes harder to prove. If we weaken the property, the induction hypothesis gives us less information, but the conclusion is easier to prove. Fortunately, there are easy cases in which the nature of the mutually recursive judgments suggested a generalization.

Finding a counterexample greatly varies in difficulty. Mostly, in this course, counterexamples only arise if there are glaring deficiencies in the inductive definitions, or rather obvious failure of properties such as type safety. In other cases it might require a very deep insight into the nature of a particular inductive definition and cannot be gleaned directly from a failed proof attempt. An example of a difficult counterexample is given by the extra credit Question 2.2 in Assignment 1 of this course. The conjecture might be that every tautology is a theorem. However, there is very little in the statement of this theorem or in the definition of *tautology* and *theorem* which would suggest means to either prove or refute it.

Three pitfalls to avoid. The difficulty with inductive proofs is that one is often blinded by the fact that the proposed conjecture is true. Similarly, if set up correctly, it will be true that in each case the induction hypothesis does in fact imply the desired conclusion, but the induction hypothesis may not be strong enough to prove it. So you must avoid the temptation to declare something as "clearly true" and prove it instead.

The second kind of mistake in an inductive proof that one often encounters is a confusion about the direction of an inference rule. If you reason backwards from what you are trying to prove, you are thinking about the rules bottom up: "*If I only could prove J_1 then I could conclude J_2, because I have an inference rule with premise J_1 and conclusion J_2.*" Nonetheless, when you write down the proof in the end you must use the rule in the proper direction. If you reason forward from your assumptions using the inference rules top-down then no confusion can arise. The only exception is the proof principle of inversion, which you can *only* employ if (a) you have established that a derivation of a given judgment J exists, and (b) you consider all possible inference rules whose conclusion matches J. We will see examples of this form of reasoning later in the course. In no other case can you use an inference rule "backwards".

The third mistake to avoid is to apply the induction hypothesis to a derivation that is not a subderivation of the one you are given. Such reasoning is circular and unsound. You must always verify that when you claim something follows by induction hypothesis, it is in fact legal to apply it!

How much to write down. Finally, a word on the level of detail in the proofs we give and the proofs we expect you to provide in the homework assignments. The proofs in this handout are quite pedantic, but we ask you to be just as pedantic unless otherwise specified. In particular, you *must* show any lemmas you are using, and you *must* show the generalized induction hypothesis in an inductive proof (if you need a generalization). You also *must* consider all the cases and *justify each line* carefully. As we gain a certain facility with such proofs, we may relax these requirements once we are certain you know how to fill in the steps that one might omit, for example, in a research paper.

Specifications vs. implementations. The grammar of our language of properly balanced parentheses (and also its formulation as an inductive definition) must be seen as a *specification*. That is, we define a language of strings (in the case of the grammar) or the judgment $s\ M$ (in the case of a judgment), but we do not immediately provide an *implementation*. In this case, such an implementation would be an algorithm for recognizing if a given string is a properly balanced string of parentheses. Ambiguity in the grammar, as noted in class, is one obstacle to deriving a parser from the specification of a grammar. In general, there are large classes of languages (including those specified by a context-free grammars) for which we can uniformly generate a parser from a grammar. Here, we will pursue a different direction, namely writing a parser for this specific language and proving that it is correct.

Interpreting inference rules as algorithms. To implement a parser, one would normally pick a programming language and simply write a program. However, then we would be faced with the problem of proving the correctness of that program, which depends on the details of the definition of the underlying implementation language.

Here we exploit instead that it is also possible to present some algorithms in the form of inference rules. Performing the algorithm corresponds to the search for a deduction of a judgment, as we will see shortly below. In programming language terminology this approach is called *logic programming*.

But first we have to decide on an algorithm for recognizing if a given string consists of properly matched parentheses. The informal idea of the parsing process for matching parentheses is quite straightforward: we keep an integer counter, initialized to zero, and increment it when we see an opening parenthesis and decrement it when we see a closing parenthesis. We need to check two conditions: (a) the counter never becomes negative (otherwise there would be too many closing parentheses) and (b) the counter is zero at the end (otherwise there would be unmatched open parentheses).

The process of parsing then corresponds to the bottom-construction of a derivation for a judgment

$$k \rhd s$$

which means that s is a valid string with respect to counter k. More specifically, s is a valid string, given that we have already seen k left parentheses that have not yet been matched by right parentheses. We assume that $k \geq 0$ is an integer. The symbol $\rhd$ has no special meaning here—it is simply used to separate the integer k from the string s. We now develop the rules for this two-place (binary) judgment.

First, if the string s is empty then we accept if the counter is 0. This corresponds to condition (b) mentioned above.

$$\frac{}{0 \rhd \varepsilon}\ \rhd_1$$

Second, if the string s starts with an opening parenthesis, we increment the counter by 1. A less operational reading is: if s is a valid string with respect to $k+1$, then $\texttt{(}s$ is a valid string in stack k.

$$\frac{k+1 \rhd s}{k \rhd \texttt{(}s}\ \rhd_2$$

Finally, if we see a closing parenthesis at the beginning of the string, then we subtract one from the counter. It is important to check that the counter remains non-negative; otherwise we might be accepting incorrectly formed strings. A less operational reading is: if s is a valid string with counter $k > 0$ then $\texttt{)}\,s$ is a valid string with counter $k-1$.

$$\frac{k-1 \rhd s \quad (k > 0)}{k \rhd \texttt{)}\,s}\ \rhd_3$$

Since these are all the rules, the bottom-up construction of a derivation will get stuck if the string s begins with a closing parentheses and k is zero. That is, there is no rule with which we could infer $0 \rhd \texttt{)}\,s$, no matter what s is. This corresponds to condition (a) mentioned at the beginning of this discussion.

It is easy to see that this parser is inherently unambiguous. That is, when we start to construct a derivation of $0 \rhd s$ in order to parse s, then at each stage there is at most one rule that can be applied, depending on whether s is empty (rule $\rhd_1$), starts with an opening parenthesis (rule $\rhd_2$), or starts with a closing parenthesis (rule $\rhd_3$). Therefore, we can think of the judgment as describing a deterministic algorithm for parsing a string.

This judgment can be related to a *push-down automaton*. Instead of a counter k, we would have a stack $\texttt{(}\,\texttt{(}\cdots\texttt{(}$ consisting of k opening parentheses. It is easy to rewrite the rules above into this form. As an aside, it turns out that every context-free grammar can be accepted by a (possibly nondeterministic) pushdown automaton, although the general construction of a pushdown automaton from a context-free grammar is more complex than in this particular example.

But does the judgment above really accept the language of properly balanced parentheses? We would like to prove that $s\ M$ if and only if $0 \rhd s$. As usual, we break this up into two separate lemmas, one for each direction.

For the first direction, we need one further lemma that captures the essence of the left-to-right processing of the input string and the use of k as a counter of unmatched open parentheses. This lemma would typically be conjectured (and then proven) only in reaction to a gap in the proof of the main theorem, but when written up it should be presented in the opposite order.

Lemma 6 (Stack)
If $k_1 \rhd s_1$ *and* $k_2 \rhd s_2$ *then* $k_1 + k_2 \rhd s_1\, s_2$

Proof: By rule induction on the derivation of $k_1 \rhd s_1$.

(Rule $\rhd_1$) Then $k_1 = 0$ and $s_1 = \varepsilon$.

$k_2 \rhd s_2$	Assumption
$k_1 + k_2 \rhd s_1\, s_2$	Since $k_1 = 0$ and $s_1 = \varepsilon$

(Rule $\rhd_2$) Then $s_1 = \texttt{(}s_1'$.

$k_1 + 1 \rhd s_1'$	Subderivation
$k_2 \rhd s_2$	Assumption
$k_1 + k_2 + 1 \rhd s_1'\, s_2$	By i.h.
$k_1 + k_2 \rhd \texttt{(}s_1'\, s_2$	By rule ($\rhd_2$)

(Rule $\rhd_3$) Then $s_1 = \texttt{)}\,s_1'$ and $k_1 > 0$.

$k_1 - 1 \rhd s_1'$	Subderivation
$k_2 \rhd s_2$	Assumption
$k_1 + k_2 - 1 \rhd s_1'\, s_2$	By i.h.
$k_1 + k_2 > 0$	Since $k_1 > 0$, $k_2 \geq 0$
$k_1 + k_2 \rhd \texttt{)}\,s_1'\, s_2$	By rule ($\rhd_3$)

■

Now we can prove the first direction of the correctness theorem for the parser.

Lemma 7
If $s\ M$ *then* $0 \rhd s$.

Proof: By rule induction on the derivation of $s\ M$.

(Rule m_1) Then $s = \varepsilon$.

$0 \rhd \varepsilon$	By rule ($\rhd_1$)

(Rule m_2) Then $s = \texttt{(}s'\texttt{)}$.

$s'\ M$	Subderivation
$0 \rhd s'$	By i.h.
$0 \rhd \varepsilon$	By rule ($\rhd_1$)
$1 \rhd \texttt{)}$	By rule ($\rhd_3$)
$1 \rhd s'\texttt{)}$	By Lemma 6
$0 \rhd \texttt{(}s'\texttt{)}$	By rule ($\rhd_2$)

(Rule m_3) Then $s = s_1\, s_2$.

$s_1\ M$	Subderivation
$0 \rhd s_1$	By i.h.
$s_2\ M$	Subderivation
$0 \rhd s_2$	By i.h.
$0 \rhd s_1\, s_2$	By Lemma 6

■

In order to prove the other direction (if $0 \rhd s$ then $s\ M$) we first generalize to:

$$\text{If } k \rhd s \text{ then } \underbrace{\texttt{(} \cdots \texttt{(}}_{k}\ s\ M.$$

This proof (which is left to the reader) requires another lemma, this time about the M judgment. Finally, putting the two directions together proves the correctness of our parser.

Summary. In this lecture, we introduced the concept of an *inductive definition* of a *judgment*, presented in the form of *inference rules*. As examples, we used inductive presentations of grammars and showed how to prove their equivalence via *rule induction*. We also sketched how algorithms can be presented via inference rules, using a parsing algorithm as an example. This form of presentation for algorithms, where computation is modeled by search for a deduction, is called *logic programming*,