Lecture Notes 2
Inductive Definitions

Carlo Angiuli

B522: PL Foundations
January 14, 2026

Our first order of business will be to define the set of (valid) expressions of a language, introducing us to the notion of an *inductive definition* by a collection of *inference rules*. This lecture will introduce inference rules, what they mean, how to reason about them, and a few applications. The material may seem technical at first, but we will see many more examples of these concepts in the coming lectures.

Abstract syntax and inductive definitions are covered in Chapters 1 and 2 of Harper [Har16]. Note that Harper's presentation is quite different from mine, but the results are similar.

1 BNF grammars

The most concise way to define a set of expressions, and one you may have seen before, is to write a formal grammar in *Backus–Naur form* (BNF). In C311/B521 you might have written down the syntax of a simple "programming language" of booleans as follows:

$$\textit{Expressions} \quad e ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{(not}\ e\texttt{)} \mid \texttt{(if}\ e\ e\ e\texttt{)}$$

In this class we are not using Racket, so we will instead write:

$$\textit{Expressions} \quad e ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{not}(e) \mid \texttt{if}(e, e, e)$$

Remark 2.1. In practice, one often wants a friendlier notation for `if`-expressions, such as `if` e `then` e `else` e. For now we are talking about a mathematical or *abstract* conception of the syntax of a programming language, divorced from questions such as how to type it into a computer. Soon enough we will allow ourselves to write friendlier *concrete* syntax for our own sake, with the understanding that it is (in this class) a shorthand for abstract syntax.

Here e ranges over the set of all possible expressions, which are `true`, `false`, `not`(e) where e is an expression, and `if`(e, e, e) where each of these es is a (possibly different) expression. (Sometimes people write `if`(e, e', e'') to make it clearer that these es are not required to be the same.)

Remark 2.2. More precisely, we are defining a single grammatical category of *expressions*, corresponding to the *nonterminal* e on the left; there are two *terminal* symbols, namely `true` and `false`, and the vertical bars indicate alternatives.

Exercise 2.3. Are the following valid expressions?

- `true` — yes
- `not(true)` — yes
- `0` — no
- `if(true, false)` — no
- `not` — no(!)
- `if(true, false, 0)` — no
- `if(not(true), false, true)` — yes

Note that the clauses of the grammar not only indicate what *is* a valid term but also what *isn't*: everything else.

Definition 2.4. e is a valid expression if and only if:

- $e =$ `true` or
- $e =$ `false` or
- $e =$ `not`(e') where e' is a valid expression or
- $e =$ `if`(e_1, e_2, e_3) where e_1, e_2, e_3 are valid expressions.

The "only if" here will be crucial to our ability to reason about the properties of valid expressions.

Remark 2.5. The HtDP-heads among you may imagine something like the following data definition:

```
; An Exp is one of:
; - true
; - false
; - (make-not Exp)
; - (make-if Exp Exp Exp)
```

2 Inference rules

To formally explain why `if(not(true), false, true)` is a valid expression, we might argue as follows:

- `if(not(true), false, true)` is an expression because
  - `not(true)` is an expression because
    - `true` is an expression
  - `false` is an expression
  - `true` is an expression

The outermost step of the argument is an instance of the general fact that $\mathtt{if}(e_1, e_2, e_3)$ is an expression if e_1 is an expression, e_2 is an expression, and e_3 is an expression; in this case e is `not(true)` and so seeing that it is an expression relies in turn on the general fact that $\mathtt{not}(e)$ is an expression if e is an expression, and that `true` is an expression.

Let us introduce some terminology and notation to simplify this argument. We will write

$$e\ \mathsf{exp}$$

for the assertion that e is an expression. We call exp a *judgment*, in the sense that this assertion "judges" that e is indeed an expression. (We can also call it an *assertion* or a *predicate*.)

There are four ways to judge e to be an expression. One is that $\mathtt{not}(e)$ is an expression if e is an expression. We write this as an *inference rule* with one *premise* (e exp) and one *conclusion* ($\mathtt{not}(e)$ exp).

$$\frac{e\ \mathsf{exp}}{\mathtt{not}(e)\ \mathsf{exp}}$$

Two of the remaining three inference rules have no premises, while the final one has three premises.

$$\frac{}{\mathtt{true}\ \mathsf{exp}} \qquad \frac{}{\mathtt{false}\ \mathsf{exp}} \qquad \frac{e_1\ \mathsf{exp} \qquad e_2\ \mathsf{exp} \qquad e_3\ \mathsf{exp}}{\mathtt{if}(e_1, e_2, e_3)\ \mathsf{exp}}$$

Remark 2.6. An inference rule with zero premises is called an *axiom*.

One benefit of inference rule notation is that we can chain these rules together:

$$\dfrac{\dfrac{\dfrac{}{\texttt{true}~\mathsf{exp}}}{\texttt{not(true)}~\mathsf{exp}} \qquad \dfrac{}{\texttt{false}~\mathsf{exp}} \qquad \dfrac{}{\texttt{true}~\mathsf{exp}}}{\texttt{if(not(true), false, true)}~\mathsf{exp}}$$

Chaining together inference rules in this way forms a *derivation tree* whose root (at the bottom) can be concluded from its leaves/premises (at the top). If every leaf is an axiom then there are no remaining premises, the conclusion holds unconditionally, and we say that the derivation is *closed.*

Exercise 2.7. Write a (closed) derivation of `not(not(true))` exp.

Sometimes we will want to consider an arbitrary derivation tree $\mathcal{D}$ ending in a particular judgment, such as `not(true)` exp. We depict such situations as follows:

$$\begin{array}{c}\mathcal{D}\\ \bigtriangledown \\ \texttt{not(true)}~\mathsf{exp}\end{array}$$

Remark 2.8. Note that we can instantiate e, e_1, e_2, e_3 in the rules for `not` and `if` with any valid expression. In that way, those rules really stand for an infinite family of rules of the form "if *[blank]* is an expression then `not(`*[blank]*`)` is an expression," whereas the rules for `true` and `false` really are just singular rules.

Once again, crucially, we take the four inference rules above as specifying not only when something *can* be judged to be an expression but also when something *cannot.*

Definition 2.9. The judgment e exp holds if and only if there is a closed derivation tree with conclusion e exp, built only out of the rules:

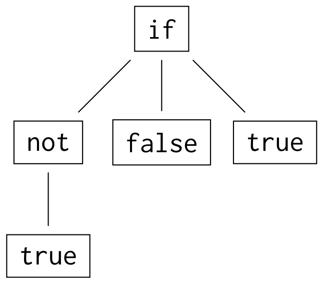
$$\dfrac{}{\texttt{true}~\mathsf{exp}} \qquad \dfrac{}{\texttt{false}~\mathsf{exp}} \qquad \dfrac{e~\mathsf{exp}}{\texttt{not}(e)~\mathsf{exp}} \qquad \dfrac{e_1~\mathsf{exp} \quad e_2~\mathsf{exp} \quad e_3~\mathsf{exp}}{\texttt{if}(e_1, e_2, e_3)~\mathsf{exp}}$$

This is essentially just a rephrasing of Definition 2.4.

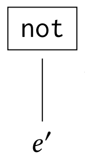
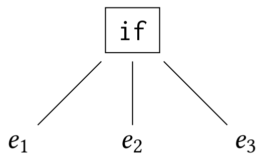
Remark 2.10. We can read inference rules from top to bottom: "if e exp then `not(`e`)` exp." But we can also read them from bottom to top: "to show that `not(`e`)` exp, we can show that e exp." But importantly, inference rules are more than just random implications: they are always the *defining clauses of a judgment.*

2.1 Aside: abstract syntax trees

If we turn the derivation of `if(not(true)`, `false`, `true)` exp upside-down, we can see it as an *abstract syntax tree*, where each node corresponds to the rule being invoked (equivalently, the topmost constructor of the expression), and its child trees correspond to the list of immediate subexpressions.



Definition 2.11. e is a valid abstract syntax tree if and only if:

- $e = \boxed{\texttt{true}}$ or
- $e = \boxed{\texttt{false}}$ or
- $e =$  where e' is a valid abstract syntax tree or
- $e =$  where e_1, e_2, e_3 are valid abstract syntax trees.

Again, this is essentially just a rephrasing of Definitions 2.4 and 2.9.

Remark 2.12. These trees show up in actual implementations of programming languages, sometimes called *abstract syntax trees* or *parse trees*, as the data structures generated by parsers.

2.2 More inductive structures

Inference rules are a powerful tool that will allow us to define a great many "inductive" structures, structures that are defined by a collection of (potentially self-referential) clauses and no others.

Example 2.13. We define the natural numbers $\mathbb{N}$ as follows:

$$\textit{Natural numbers} \quad n ::= \quad \texttt{zero} \mid \texttt{suc}(n)$$

where the "successor" of n represents $n + 1$. Equivalently, the judgment n nat is defined by the collection of inference rules:

$$\frac{}{\texttt{zero}\ \mathsf{nat}} \qquad \frac{n\ \mathsf{nat}}{\texttt{suc}(n)\ \mathsf{nat}}$$

Example 2.14. We could add natural numbers to our programming language:

$$\begin{array}{lll} \textit{Expressions} & e ::= & \texttt{true} \mid \texttt{false} \mid \texttt{not}(e) \mid \texttt{if}(e, e, e) \\ & \mid & \texttt{zero} \mid \texttt{suc}(e) \mid \texttt{pred}(e) \mid \texttt{zero?}(e) \end{array}$$

Equivalently,

$$\frac{}{\texttt{true}\ \mathsf{exp}} \qquad \frac{}{\texttt{false}\ \mathsf{exp}} \qquad \frac{e\ \mathsf{exp}}{\texttt{not}(e)\ \mathsf{exp}} \qquad \frac{e_1\ \mathsf{exp} \qquad e_2\ \mathsf{exp} \qquad e_3\ \mathsf{exp}}{\texttt{if}(e_1, e_2, e_3)\ \mathsf{exp}}$$

$$\frac{}{\texttt{zero}\ \mathsf{exp}} \qquad \frac{e\ \mathsf{exp}}{\texttt{suc}(e)\ \mathsf{exp}} \qquad \frac{e\ \mathsf{exp}}{\texttt{pred}(e)\ \mathsf{exp}} \qquad \frac{e\ \mathsf{exp}}{\texttt{zero?}(e)\ \mathsf{exp}}$$

We can also define multiple grammatical categories / judgments simultaneously with reference to one another.

Example 2.15. We define lists of natural numbers as follows:

$$\begin{array}{lll} \textit{Natural numbers} & n ::= & \texttt{zero} \mid \texttt{suc}(n) \\ \textit{Lists of naturals} & \ell ::= & \texttt{empty} \mid \texttt{cons}(n, \ell) \end{array}$$

Equivalently,

$$\frac{}{\texttt{zero}\ \mathsf{nat}} \qquad \frac{n\ \mathsf{nat}}{\texttt{suc}(n)\ \mathsf{nat}} \qquad \frac{}{\texttt{empty}\ \mathsf{natlist}} \qquad \frac{n\ \mathsf{nat} \qquad \ell\ \mathsf{natlist}}{\texttt{cons}(n, \ell)\ \mathsf{natlist}}$$

Example 2.16. We define even and odd natural numbers as follows:

$$\begin{array}{lll} \textit{Evens} & e ::= & \texttt{zero} \mid \texttt{suc}(o) \\ \textit{Odds} & o ::= & \texttt{suc}(e) \end{array}$$

Equivalently,

$$\frac{}{\texttt{zero}\ \mathsf{even}} \qquad \frac{o\ \mathsf{odd}}{\texttt{suc}(o)\ \mathsf{even}} \qquad \frac{e\ \mathsf{even}}{\texttt{suc}(e)\ \mathsf{odd}}$$

Remark 2.17. BNF grammars will not be able to fully capture most of the languages we consider throughout the semester, because they have an infinite number of grammatical categories (types) and/or binding (variables); for these features we will need to use inference rules. However, BNF grammars will remain a convenient way to summarize some systems of inference rules.

So far all of our judgments have been *unary*: assertions about a single expression (often, that it is well-formed). We can also define *binary* judgments that make a joint assertion about two things.

Example 2.18. The binary judgment

$$n \ \mathsf{doubledIs}\ n'$$

is defined by the following inference rules:

$$\frac{}{\mathsf{zero}\ \mathsf{doubledIs}\ \mathsf{zero}} \qquad \frac{n\ \mathsf{doubledIs}\ n'}{\mathsf{suc}(n)\ \mathsf{doubledIs}\ \mathsf{suc}(\mathsf{suc}(n'))}$$

Exercise 2.19. There is exactly one concrete n for which we can derive

$$\mathsf{suc}(\mathsf{zero})\ \mathsf{doubledIs}\ n$$

What is that n? Write the derivation of the above judgment.

2.3 Derivable and admissible properties

Lemma 2.20. *If* n even *then* $\mathsf{suc}(\mathsf{suc}(n))$ even.

Proof. Suppose we have a derivation $\mathcal{D}$ of n even. Then we can add two more rules to the bottom of $\mathcal{D}$ to form a derivation of $\mathsf{suc}(\mathsf{suc}(n))$ even as required:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\bigtriangledown}}{\begin{array}{c} n\ \mathsf{even}\end{array}}}{}$$

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{D}\\ \bigtriangledown \\ n\ \mathsf{even}\end{array}}{\mathsf{suc}(n)\ \mathsf{odd}}}{\mathsf{suc}(\mathsf{suc}(n))\ \mathsf{even}}$$

□

So far we have used the "if" direction of a number of inductive definitions (as in Definitions 2.4, 2.9 and 2.11) to show that certain judgments hold, and we have used the "only if" direction to show that certain judgments do *not* hold (e.g., `0` is not an expression). We can also use the "only if" direction to prove positive properties of judgments.

Lemma 2.21 (Inversion for suc)**.** *If* $\mathsf{suc}(m)$ nat *then* m nat.

Proof. The judgment n nat holds *only if* there is a derivation of n nat built out of repeated applications of the two rules

$$\frac{}{\mathsf{zero}\ \mathsf{nat}} \qquad\qquad \frac{n\ \mathsf{nat}}{\mathsf{suc}(n)\ \mathsf{nat}}$$

In particular, any derivation of $\mathsf{suc}(m)$ nat must end with an application of one of these two rules. It cannot end with an application of the first rule, because then it would be a derivation of $\mathtt{zero}$ nat. So it must end with the second rule applied to a complete derivation of its premise m nat. From this we conclude m nat. ◻

Remark 2.22. The above argument is an instance of a common reasoning pattern called *rule induction*, which we will make more precise shortly.

Remark 2.23. Whereas the second rule says that m nat is a *sufficient* condition for concluding that $\mathsf{suc}(m)$ nat, Lemma 2.21 states that it is in fact a *necessary* condition: that rule is the *only* way to conclude that $\mathsf{suc}(m)$ nat. In other words, we can "run the rule backwards," which is why it's called an *inversion lemma.*[5]

Lemmas 2.20 and 2.21 are a good illustration of the important distinction between *derivable* and *admissible* properties of inductively-defined sets.

Derivability Although Lemma 2.20 ("n even $\implies$ $\mathsf{suc}(\mathsf{suc}(n))$ even") is *not* one of the defining rules of the even judgment, we can prove it by chaining together basic inference rules without any case analysis of what n or the derivation of n even might be. For this reason, Lemma 2.20 is a very robust property of evenness: it will continue to hold even if we add more basic inference rules to the system, since it only relies on the presence of certain rules and not the absence of others. We say that n even $\implies$ $\mathsf{suc}(\mathsf{suc}(n))$ even is *derivable.*

Admissibility In contrast, consider Lemma 2.21 ($\mathsf{suc}(m)$ nat $\implies$ m nat), which is also not one of the defining rules of natural numbers, but whose proof requires case analysis on all the possible derivations of $\mathsf{suc}(m)$. For this reason, Lemma 2.21 is a much less robust property which *can* be disrupted by the addition of inference rules such as:

$$\frac{\ell\ \mathsf{natlist}}{\mathsf{suc}(\ell)\ \mathsf{nat}}$$

in whose presence $\mathtt{suc(empty)}$ nat would hold but $\mathtt{empty}$ nat would not. We say that the property $\mathsf{suc}(m)$ nat $\implies$ m nat is *admissible.*

[5]Some students may have previously encountered inversion in the film *Tenet* (2020).

Exercise 2.24. Propose an inference rule that could be added to the nat judgment such that Lemma 2.21 continues to hold. (The resulting nat judgment doesn't need to be a sensible definition of natural numbers.)

Definition 2.25. An *admissible rule* is any implication of judgments that can be proven, possibly by case analysis; a *derivable rule* is any implication of judgments that can be proven by a uniform combination of inference rules.

Note that all derivable rules are admissible, but not vice versa. (However, when we say that a principle is admissible, we will usually mean that it is not derivable.)

3 Least sets

Before giving any more examples of rule induction, I want to pause and derive it mathematically from the "if and only if" characterization of inductive structures.

Let us write Exp for the set of valid expressions: the set of e for which e exp holds. Recall that by Definition 2.9, $e \in \mathsf{Exp}$ if and only if we have a derivation tree with conclusion e exp built only out of the following rules:

$$\frac{}{\texttt{true}\ \mathsf{exp}} \qquad \frac{}{\texttt{false}\ \mathsf{exp}} \qquad \frac{e\ \mathsf{exp}}{\texttt{not}(e)\ \mathsf{exp}} \qquad \frac{e_1\ \mathsf{exp} \qquad e_2\ \mathsf{exp} \qquad e_3\ \mathsf{exp}}{\texttt{if}(e_1, e_2, e_3)\ \mathsf{exp}}$$

Let us refer to these four rules as E.

Definition 2.26. A set X is *E-closed* if the following four properties hold:

- $\texttt{true} \in X$,
- $\texttt{false} \in X$,
- $\texttt{not}(x) \in X$ whenever $x \in X$, and
- $\texttt{if}(x_1, x_2, x_3)$ whenever $x_1, x_2, x_3 \in X$.

Theorem 2.27. *The set of expressions* Exp *is the* least E-closed set*; that is, (1)* Exp *is E-closed, and (2) for any E-closed set X, we have* $\mathsf{Exp} \subseteq X$*.*

Proof.

1. First we prove that Exp is E-closed. There are derivations of `true` exp and `false` exp, so by definition we have $\texttt{true}, \texttt{false} \in \mathsf{Exp}$. Using the `not` rule, whenever we have a derivation of e exp we can build a derivation of $\texttt{not}(e)$ exp. Finally, using the `if` rule, whenever we have derivations of e_1 exp and e_2 exp and e_3 exp we can build a derivation of $\texttt{if}(e_1, e_2, e_3)$.

2. Suppose X is E-closed and we have a derivation $\mathcal{D}$ of e exp; we must show that $e \in X$. We proceed by strong induction on the height of the tree $\mathcal{D}$. If $\mathcal{D}$ has height zero, then it must be a derivation of `true` exp or `false` exp; in either case we have `true`, `false` $\in X$ by E-closure.

   If instead $\mathcal{D}$ has height $h + 1$, there are two possibilities, the first being:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \bigtriangledown \\ e \ \mathsf{exp}\end{array}}{\mathtt{not}(e) \ \mathsf{exp}}$$

   where $\mathcal{D}'$ has height h. In this case, by the inductive hypothesis applied to $\mathcal{D}'$, we have $e \in X$; by E-closure, this implies `not`(e) $\in X$ as required. The other possibility is:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \bigtriangledown \\ e_1 \ \mathsf{exp}\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \bigtriangledown \\ e_2 \ \mathsf{exp}\end{array} \qquad \begin{array}{c}\mathcal{D}_3 \\ \bigtriangledown \\ e_3 \ \mathsf{exp}\end{array}}{\mathtt{if}(e_1, e_2, e_3) \ \mathsf{exp}}$$

   where $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ have heights at most h. The (strong) inductive hypotheses imply $e_1, e_2, e_3 \in X$, and once again `if`(e_1, e_2, e_3) $\in X$ by E-closure. □

Example 2.28. For a concrete example of the above proof, suppose that X is E-closed and show that `if(not(true), false, true)` $\in X$ by converting each inference rule in the derivation below to an application of E-closure:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{true} \ \mathsf{exp}}}{\mathtt{not(true)} \ \mathsf{exp}} \qquad \dfrac{}{\mathtt{false} \ \mathsf{exp}} \qquad \dfrac{}{\mathtt{true} \ \mathsf{exp}}}{\mathtt{if(not(true), false, true)} \ \mathsf{exp}}$$

Remark 2.29. The E-closure of Exp corresponds to the "if" direction of our definition: if we can build a derivation out of the rules of E, then we have a valid expression. The leastness corresponds to the "only if" direction: Exp contains only things built out of the rules of E. If Exp contained additional expression(s) *not* built out of the rules, those expressions would not be in every E-closed set.

Let's look at a more familiar example next.

Definition 2.30. A set X is *N-closed* if $0 \in X$ and $n \in X \implies n + 1 \in X$.

Definition 2.31. The natural numbers $\mathbb{N}$ are the least N-closed set.

Remark 2.32. Examples of other N-closed sets include $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N} \cup \{-1\}$, …all of which satisfy the necessary closure conditions but include superfluous elements.

The characterization of $\mathbb{N}$ as the least N-closed set implies the principle of mathematical induction.

Theorem 2.33 (Mathematical induction)**.** *To prove that $P(n)$ holds for all $n \in \mathbb{N}$, it suffices to show that $P(0)$ and $\forall n \in \mathbb{N}.P(n) \implies P(n+1)$.*

Proof. Define $\Sigma_P = \{n \in \mathbb{N} \mid P(n)\}$. Clearly $\Sigma_P \subseteq \mathbb{N}$ by construction. The hypotheses imply that Σ_P is N-closed, so by Definition 2.31 we have $\mathbb{N} \subseteq \Sigma_P$. Hence $\mathbb{N} = \Sigma_P$ and thus $P(n)$ for all n. □

Similarly, our characterization of Exp as the least E-closed set implies a principle of *rule induction*: to prove a property P for all expressions, we must show that the subset of expressions satisfying P is E-closed. That is, the property P is preserved by every inference rule.

Theorem 2.34 (Rule induction for Exp)**.** *To prove that a property $P(e)$ holds for all $e \in \mathsf{Exp}$, it suffices to show that:*

- $P(\texttt{true})$,
- $P(\texttt{false})$,
- *for every e, if $P(e)$ then $P(\texttt{not}(e))$, and*
- *for every e_1, e_2, e_3, if $P(e_1)$, $P(e_2)$, and $P(e_3)$, then $P(\texttt{if}(e_1, e_2, e_3))$.*

Proof. Define $\Sigma_P = \{e \in \mathsf{Exp} \mid P(e)\}$. Clearly $\Sigma_P \subseteq \mathsf{Exp}$ by construction. The hypotheses imply that Σ_P is E-closed, so by Theorem 2.27 we have $\mathsf{Exp} \subseteq \Sigma_P$. Hence $\mathsf{Exp} = \Sigma_P$ and thus $P(e)$ for all $e \in \mathsf{Exp}$. □

example where the premises are larger or it's otherwise not syntax-directed, so induction on the derivation isn't confused with induction on the term.

4 Rule induction

Theorem 2.35 (Rule induction for nat)**.** *To prove that a property $P(n)$ holds for all n nat, it suffices to show that:*

- $P(\texttt{zero})$ *and*

- *for every n, if $P(n)$ then $P(\mathsf{suc}(n))$.*

Remark 2.36. Theorems 2.34 and 2.35 demonstrate that rule induction is in some sense a strict generalization of mathematical induction. On the other hand, as demonstrated in our proof of Theorem 2.27, rule induction is not actually provable without appeal to mathematical induction (or similar principles).

Lemma 2.37. *If n nat then there exists some n' such that n $\mathsf{doubledIs}$ n'.*

Proof. We prove $P(n)$ = "there exists some n' such that n $\mathsf{doubledIs}$ n'" by rule induction. We must show that P is closed under the rules for the nat judgment, of which there are two.

- The first rule is:

$$\frac{}{\mathsf{zero\ nat}}$$

  We must show that P holds in this case, i.e. that $P(\mathsf{zero})$ holds, i.e. that there exists some n' such that zero $\mathsf{doubledIs}$ n'. We choose $n' = \mathsf{zero}$, because:

$$\frac{}{\mathsf{zero\ doubledIs\ zero}}$$

- The second rule is:

$$\frac{n\ \mathsf{nat}}{\mathsf{suc}(n)\ \mathsf{nat}}$$

  We must show that if the premise satisfies P (this is called our *inductive hypothesis*), then the conclusion satisfies P. That is, we must show that for all n, if $P(n)$ holds then $P(\mathsf{suc}(n))$ holds. Unfolding the definition of P, our inductive hypothesis $P(n)$ is that there exists n' such that n $\mathsf{doubledIs}$ n'. We must show that $P(\mathsf{suc}(n))$ holds, which is to say that there exists n'' such that $\mathsf{suc}(n)$ $\mathsf{doubledIs}$ n''. We choose $n'' = \mathsf{suc}(\mathsf{suc}(n'))$ because:

$$\frac{n\ \mathsf{doubledIs}\ n'}{\mathsf{suc}(n)\ \mathsf{doubledIs}\ \mathsf{suc}(\mathsf{suc}(n'))}$$ □

Inversion Lemma 2.21 also follows from rule induction, for the property $P(n)$ = "n nat, and if n is of the form $\mathsf{suc}(n')$ for some n', then n' nat."

Theorem 2.38 (Rule induction for $\mathsf{doubledIs}$)**.** *To prove that a property $P(n, n')$ holds for all n $\mathsf{doubledIs}$ n', it suffices to show that:*

- $P(\mathsf{zero}, \mathsf{zero})$ *and*
- *for all* n, n', *if* $P(n, n')$ *then* $P(\mathsf{suc}(n), \mathsf{suc}(\mathsf{suc}(n')))$.

Lemma 2.39. *If* n doubledIs n' *then* n' even.

Proof. We prove $P(n, n') =$ "n' even" by rule induction. There are two cases:

- Case 1:

$$\frac{}{\mathsf{zero\ doubledIs\ zero}}$$

  We must prove that the conclusion satisfies P, i.e. $P(\mathsf{zero}, \mathsf{zero})$, or that zero even. This is an axiom.

- Case 2:

$$\frac{n \ \mathsf{doubledIs}\ n'}{\mathsf{suc}(n)\ \mathsf{doubledIs}\ \mathsf{suc}(\mathsf{suc}(n'))}$$

  We must prove that P is closed under this rule: in other words, if $P(n, n')$ holds then $P(\mathsf{suc}(n), \mathsf{suc}(\mathsf{suc}(n')))$ holds. Our inductive hypothesis $P(n, n')$ is that n' even, and we must prove that $\mathsf{suc}(\mathsf{suc}(n'))$ even. We already proved this in Lemma 2.20. □

Lemma 2.40. *If* n doubledIs n' *and* n doubledIs n'' *then* $n' = n''$.

Proof. We prove $P(n, n') =$ "for all n'' for which n doubledIs n'', we have $n' = n''$" by rule induction (on n doubledIs n'). There are two ways that the derivation of n doubledIs n' can end:

- Case 1:

$$\frac{}{\mathsf{zero\ doubledIs\ zero}}$$

  Show that $P(\mathsf{zero}, \mathsf{zero})$: if zero doubledIs n'', then $\mathsf{zero} = n''$. This follows by inversion, because the only rule that could derive such a thing is

$$\frac{}{\mathsf{zero\ doubledIs\ zero}}$$

  Therefore $n'' = \mathsf{zero}$, which is what we wanted to show.

- Case 2:
$$\frac{n\ \mathsf{doubledIs}\ n'}{\mathsf{suc}(n)\ \mathsf{doubledIs}\ \mathsf{suc}(\mathsf{suc}(n'))}$$

  Prove $P(n, n')$ implies $P(\mathsf{suc}(n), \mathsf{suc}(\mathsf{suc}(n')))$. The latter property is: if $\mathsf{suc}(n)$ $\mathsf{doubledIs}$ n'', then $\mathsf{suc}(\mathsf{suc}(n')) = n''$. Our inductive hypothesis is the analogous statement about the premise: if n $\mathsf{doubledIs}$ m then $n' = m$.

  By inversion on the supposed derivation of $\mathsf{suc}(n)$ $\mathsf{doubledIs}$ n'', only one rule applies:
$$\frac{n\ \mathsf{doubledIs}\ m'}{\mathsf{suc}(n)\ \mathsf{doubledIs}\ \mathsf{suc}(\mathsf{suc}(m'))}$$

  This tells us that n'' is of the form $\mathsf{suc}(\mathsf{suc}(m'))$, so it remains to prove that $\mathsf{suc}(\mathsf{suc}(m')) = \mathsf{suc}(\mathsf{suc}(n'))$. By n $\mathsf{doubledIs}$ n' and n $\mathsf{doubledIs}$ m' and our inductive hypothesis, $m' = n'$. □

The above proof actually uses *nested* rule induction: we consider the two cases of n $\mathsf{doubledIs}$ n', and in each case, we consider the two cases of n $\mathsf{doubledIs}$ n'' (implicitly, in our appeal to inversion).

By Lemmas 2.37 and 2.40, for every n nat there is exactly one n' such that n $\mathsf{doubledIs}$ n'. In other words, doubling is in fact a *function* on natural numbers. We will often use inference rules to define functions in this way.

Theorem 2.41 (Structural recursion for nat)**.** *To define a function f on natural numbers, one must:*

- *define $f(\mathsf{zero})$, and*
- *define $f(\mathsf{suc}(n))$ for arbitrary n, given the value of $f(n)$.*

Proof. As with $\mathsf{doubledIs}$, we can define a binary judgment n $\mathsf{maps\text{-}to}$ n' by the inference rules:

$$\frac{}{\mathsf{zero}\ \mathsf{maps\text{-}to}\ \ldots} \qquad \frac{n\ \mathsf{maps\text{-}to}\ n'}{\mathsf{suc}(n)\ \mathsf{maps\text{-}to}\ (\ldots n' \ldots)}$$

The proofs of Lemmas 2.37 and 2.40 show that for any n nat there is exactly one n' for which n $\mathsf{maps\text{-}to}$ n', so that $\mathsf{maps\text{-}to}$ is the graph of a function. □

To perform rule induction on mutually-defined judgments, we need mutual induction hypotheses.

Theorem 2.42 (Rule induction for even/odd)**.** *If*

- $P(\mathsf{zero})$,
- *for all* o odd, $Q(o) \implies P(\mathsf{suc}(o))$, *and*
- *for all* e even, $P(e) \implies Q(\mathsf{suc}(e))$,

then $P(e)$ *holds for all* e even *and* $Q(o)$ *holds for all* o odd.

Theorem 2.43. *If* e even *then* e nat, *and if* o odd *then* o nat.

Proof. We prove these by mutual rule induction.

- For the rule

  $$\frac{}{\mathsf{zero}\ \text{even}}$$

  we must show zero nat, which is true by an axiom.

- For the rule

  $$\frac{o\ \text{odd}}{\mathsf{suc}(o)\ \text{even}}$$

  our inductive hypothesis is o nat and we must show $\mathsf{suc}(o)$ nat, which holds by the suc rule for nat.

- For the rule

  $$\frac{e\ \text{even}}{\mathsf{suc}(o)\ \text{odd}}$$

  our inductive hypothesis is e nat and we must show $\mathsf{suc}(e)$ nat, which holds by the suc rule for nat. □

Remark 2.44. HtDP-heads might have noticed

collection of inference rules : data definition :: rule induction : template

except that rule induction is about constructing a *proof* out of the input derivation. Structural recursion brings us full circle, using rule induction as a means of defining an ordinary function using a recursive template.

5 Evaluation

I didn't want to go a whole lecture without saying anything about programming languages. Going back to the boolean expression language from the beginning of the lecture, we can use inference rules to define a binary *evaluation* judgment, $e \Downarrow e'$, which explains how to recursively simplify any expression e.

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}} \qquad \frac{e \Downarrow \texttt{true}}{\texttt{not}(e) \Downarrow \texttt{false}} \qquad \frac{e \Downarrow \texttt{false}}{\texttt{not}(e) \Downarrow \texttt{true}}$$

$$\frac{e_1 \Downarrow \texttt{true} \qquad e_2 \Downarrow e_2'}{\texttt{if}(e_1, e_2, e_3) \Downarrow e_2'} \qquad \frac{e_1 \Downarrow \texttt{false} \qquad e_3 \Downarrow e_3'}{\texttt{if}(e_1, e_2, e_3) \Downarrow e_3'}$$