ADVANCED PROGRAMMING LANGUAGES

Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

Inductive Definitions

Before we proceed, we introduce a general scheme of inductive definitions and we study the corresponding general induction principle that derives from it. Both these notions provide important technical tools instrumental to formalize most of the core notions in the theory of programming languages we will approach along the course.

An inductive definition is a constructive way to introduce a set (often a set of tuples forming a relationship). It consists of a family of rules to *derive* judgements (i.e. assertions, statements) establishing a relationships among two or more object. Examples include equality judgement such as $M = N$, typing judgements $M : T$ (M has type T), evaluation judgements, $M \to N$ (M reduces to N), membership judgements $M \in S$ (M is a term in the set S).

Inductive rules have the following general structure:

$$\frac{J_1 \quad \cdots \quad J_k}{J}$$

J and $J_1, \ldots, J_k$ are judgements of the form being defined: the judgements $J_1, \ldots, J_k$ above the line are called *premises* of the rule, and the judgement J below the line is called its *conclusion*. A rule with no premises ($k = 0$), is an *axiom*, otherwise it is a proper *inference rule*.

An inference rule can be read as stating that the premises are sufficient conditions for the conclusion: that is to say, to show J it is enough to show $J_1, \ldots, J_k$. An axiom states that its conclusion holds unconditionally.

An inductive set of rules defines the *strongest judgement form* (i.e. the minimal relation described by the judgement form) that is *closed under the rules*. More precisely:

- To be closed under the rules means that the rules are *sufficient* to derive the judgements: a judgement J holds *if* there is a way to obtain it by composing the given rules.
- To be the strongest judgement form closed under the rules means that the rules are also *necessary* to derive the judgements : a judgement J holds *only if* there is a way to obtain it by composing the rules.

Said differently, sufficiency means that we may show that J holds by deriving it by composing the rules; necessity means that it must have been derived by composing the rules. Rule composition is formalized by the notion of derivation.

Derivations. A derivation of a judgement is a finite composition of rules that starts with axioms and end with that judgement. It can be thought of as a tree (viewed upside down) in which each node is a rule whose children are derivations of its premises.

$$\dfrac{\dfrac{J_{1,1} \quad \cdots \quad J_{1,n_1}}{J_1} \quad \cdots \quad \dfrac{J_{k,1} \quad \cdots \quad J_{k,n_k}}{J_k}}{J}$$

Then, to show that a judgement J holds it is enough to find a derivation ending with J.

To illustrate, we give three examples of inductive definitions. The definitions introduce judgment forms $J(\bullet)$ that predicate over elements of a given domain.

Inductively defined natural numbers

The set of natural numbers can be defined inductively by the following system of rules for the judgement $\bullet \in \mathrm{Nat}$

$$\dfrac{}{0 \in \mathrm{Nat}} \qquad \dfrac{n \in \mathrm{Nat}}{succ(n) \in \mathrm{Nat}}$$

We typically use *rule schemes* that use meta-variables to specify an infinite family of rules on the terms build over the meta-variables.

- The first rule states that 0 belongs to Nat.
- The second is a rule scheme that says that whenever n belongs to Nat, then $succ(n)$ also is in Nat.

Collectively, the two rules define the minimal relation described by the judgement $n \in \mathrm{Nat}$ closed under the inductive rules, i.e., they construe Nat as the minimal set of terms n such that $n \in \mathrm{Nat}$ is closed under the inductive rules. That, in turn, implies that all elements of Nat may (and must!) be constructed by composing the rules in a derivation. Thus, the numeral 2 is the result of the derivation shown below:

$$\dfrac{\dfrac{0 \in \mathrm{Nat}}{succ(0) \in \mathrm{Nat}}}{succ(succ(0)) \in \mathrm{Nat}}$$

Inductively defined λ terms

Inductive definitions are commonplace in the formalization of all aspects of programming language theory, from syntax to semantics to type theory.
BNF syntactic definitions, for instance, are nothing else than inductive definitions, even though they rely on induction only implicitly. For instance, the following is an inductive definition of the set of lambda terms (Lam) equivalent to the definition we gave in our earlier lecture with the more familiar BNF notation.

$$\frac{}{x, y, z \in \mathrm{Lam}} \qquad \frac{M \in \mathrm{Lam}}{\lambda x.M \in \mathrm{Lam}} \qquad \frac{M \in \mathrm{Lam} \quad N \in \mathrm{Lam}}{M\ N \in \mathrm{Lam}}$$

Inductively defined β-reduction

As a final example, we formalize the notion of β-reduction as a relationship between two terms. In our previous lectures we have introduced β-reduction in a rather informal way, by defining the *core* rule that formalizes the behavior of application, transforming the *redex* $(\lambda x.M)N$ into the *reduct* $[N/x]M$, and then arguing that we may reduce an expression as long as we reduce one redex of the expression.

Full β reduction. We now make this formal by defining β-reduction inductively, by the following rule system for judgements of the form $\bullet_1 \rightarrow \bullet_2$, relating two terms.

$$\frac{}{(\lambda x.M)N \rightarrow [N/x]M} \qquad \frac{M \rightarrow M'}{M\ N \rightarrow M'\ N} \qquad \frac{N \rightarrow N'}{M\ N \rightarrow M\ N'} \qquad \frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'}$$

The first axiom is the β-reduction rule we have introduced in the initial presentation of the λ-calculus. The additional rules provide us with a formal device to apply the reduction inside terms – specifically (i) inside the term in function position of an application , (ii) inside the argument of an application and, finally, (iii) inside a λ-abstraction.

The Principle of Rule Induction

The principle of rule induction is a generalization of the following, well-known principle of induction on natural numbers:

Let P be any property of natural numbers. Suppose P holds of zero, and whenever P holds of a natural number n then it holds of its successor, $n + 1$. Then P holds of every natural number.

Rule induction generalizes this principle to reason about inductively defined sets (or relations) and their properties.

The principle of rule induction states that to show that a property $P(a)$ holds whenever $J(a)$ is derivable, it is enough to show that P is closed under, or respects, the rules defining the judgment form J.

More precisely, the property P *respects* the rule

$$\frac{J(a_1) \quad \cdots \quad J(a_k)}{J(a)}$$

if $P(a)$ holds whenever $P(a_1), \ldots, P(a_k)$ do. The assumptions $P(a_1), \ldots, P(a_k)$ are called the *inductive hypotheses*, and $P(a)$ is called the *conclusion* of the inference.

Applying the principle to prove a property P for our inductively defined natural amounts to show that P respects the two rules defining the judgement $\bullet \in \mathrm{Nat}$. That, in turn, requires us to show that

1. $P(0)$ holds,
2. if $P(n)$ holds for a given (arbitrary n), so does $P(\mathrm{succ}(n))$

This is just the familiar induction principle on natural numbers.

Similarly, proving that a given property P holds of all lambda terms, amounts to show that

1. $P(x)$ holds for any variable x
2. if $P(M)$ holds for any given term M, then so does $P(\lambda x.M)$
3. if $P(M)$ and $P(N)$ both hold true for arbitrary terms M and N, then so does $P(M\ N)$

Rule induction applies as well to inductively defined relations. For instance, to prove a given property P of (full) β-reduction, we must show the following:

1. P holds of $(\lambda x.M)N \to [N/x]M$ for any two (arbitrary) terms M and N
2. if P holds of $M \to M'$ then so does of $M\ N \to M'\ N$, of $N\ M \to N\ M'$ and of $\lambda x.M \to \lambda x.M'$

Exercises

1. Expressions, defined inductively

Express the syntax of expression language with an inductive definition for the judgement $M \in$

Exp.

2. Induction and Induction Principle

Let the set of integers Q be defined inductively as follows:

$$\text{RULE1}\ \frac{}{8 \in \mathsf{Q}} \qquad \text{RULE2}\ \frac{}{5 \in \mathsf{Q}} \qquad \text{RULE3}\ \frac{a \in \mathsf{Q} \quad b \in \mathsf{Q} \quad [c = a + b + 1]}{c \in \mathsf{Q}}$$

Exercise 2.1. Answer the following:

- Of the rules above (i.e., RULE1, RULE2, and RULE3), which are axioms and which are inductive rules?
- Give a derivation showing that 11 is in the set Q.
- Give a derivation showing that 20 is in the set Q.

Exercise 2.2. Write down the inductive principle for Q. That is, if you wanted to prove that for some property P, for all a ∈ Q we have P(a), what would you need to show?

Exercise 2.3. Is 2 in the set Q? If so, give a derivation proving it. Otherwise, figure out a way to disprove it (*Hint*: in the latter case, prove some property that holds true of all elements of Q and is not true of 2).

3. Induction and reduction in λ-calculus

Recall the definition of the recursion operator

def *Y* = λf.(λx. f(x x)) (λx. f(x x))

Exercise 3.1. Let F be a normal-form term.
- exaluate (*Y* F) using applicative order
- evaluate (*Y* F) using normal order

Exercise 3.2. Assume we have an applied lambda calculus with integers, booleans and conditions and consider the following high-level definition of add:

def *add* = add = λf.λa.λb. (if a = 0 then b else f(a-1)(b+1)

Now show the the computation of (*add* 2 1) recursively using *Y*. You don't need to expand any operators except *Y* and *add*, and those only when necessary. In other words, show that (*Y add*) 2 1 = 3

Exercise 3.3. Assume we have an applied lambda calculus with integers, booleans, conditionals, etc. Consider the following higher-order function H.

H = λf. λn. if n = 1 then true else if n = 0 then false else not (f (n − 1))

- Suppose that g is the fixed point of H. What does g compute?
- Compute (Y H) with CBN semantics. What happens to the call f (n − 1)?
- Compute (Y H) 2 with CBN semantics.

CREDITS

The material on induction and rule induction are taken from Robert Harpers's book on *Practical Foundations of Programming Languages (second edition).* Cambridge University Press, 2016.