The construction of Inductive Sets

Abstract

This document is both a construction and a detailed explanation of the notion of inductive set in the context of type theory. It is a litterate program for the LaTTe proof assistant.

Contents

```
(ns fixed-points.inductive-sets
  (:refer-clojure :exclude [and or not set])

  (:require [latte.core :as latte
             :refer [definition defthm defaxiom
                     forall lambda ==>
                     assume have proof try-proof]]
            [latte.quant :as q :refer [exists]]
            [latte.prop :as p :refer [<=> and or not]]
            [latte.equal :as eq :refer [equal]]
            [latte.rel :as rel :refer [rel]]
            [latte-sets.core :as set
             :refer [set elem
                     subset seteq emptyset forall-in]]
            [latte-sets.powerset :as pset
             :refer [powerset intersections]]))
```

Introduction

In this document we discuss one possible formalization of *inductive sets*. The two main source of inspiration for this document is the book *Introduction to bisimulation and coinduction* by Davide Sangiorgi as well as the course notes by Glynn Winskel about discrete mathematics, cf. https://www.cl.cam.ac.uk/~gw104/.

The main motivation is this:

Inductive sets are omnipresent in both mathematics (e.g. Peano arithmetics, proof theory, etc.) and computer science (tree structures, recursive functions, etc.). However, the underlying principles are seldom if imprecisely documented. In particular, we will study how inductive sets can be built from set-theoretic principles.

A further motivation is to elaborate the construction of inductive sets in the context of a proof assistant based on a type theory. This way, nothing that could be judged as a detail is left to change in the development. Moreover, all the proofs are checked by the computer.

Sets defined by inductive rules

Inductively defined sets are most often described using logical rules. As an introduction to the topic, we will consider a few example of well-known inductive sets.

A first example is the set of natural number, which is often described as the *least* set satisfying the rules below.

$$\frac{}{0 \in \mathbb{N}} \qquad \forall n. \frac{n \subseteq \mathbb{N}}{succ(n) \in \mathbb{N}}$$

We shall see with all details what this definition means, formally.

Another example is the set of strings over an alphabet Σ, denoted by Σ^*.

$$\frac{}{\epsilon \in \Sigma^*} \qquad \forall x. \forall a. a \in \Sigma \implies \frac{\{x\} \subseteq \Sigma^*}{a.x \in \Sigma^*}$$

A slightly different kind of inductive definition is provided by the notion of the *transitive closure of a relation.* If we suppose a relation R (from a type T to itself), then its transitive clojure, denoted by R^+, is the *least* relation satisfying:

$$\frac{}{(a,b) \in R^+} \text{for any } (a,b) \in R \qquad \frac{(a,b),(b,c) \subseteq R^+}{(a,c) \in R^+}$$

We are now looking for a general, type-theoretic, definition of these kinds of sets of rules.

Given a type T, a rule instance on T is of the form (X, y) with X a set of T-elements and y a T-element. The intended meaning is that if X is a subset of the inductive set, then we can deduce that y is *also* an element. Hence, a rule-based definition is a relation from powersets to sets.

```
(definition rules
  "The constructor for rule sets."
  [[T :type]]
  (==> (set T) T :type))
```

Closed sets

We next introduce the notion of a R-closed set, with R a rule set as defined previously.

```
(definition closed-set
  "The set `E` is `R`-closed."
  [[T :type] [R (rules T)] [E (set T)]]
  (forall [X (set T)]
      (forall [y T]
        (==> (R X y)
             (subset T X E)
             (elem T y E)))))
```

The definition above provide a precise meaning for the intuitive understanding of the inductive rules. if a pair (X, y) is a rule instance, i.e. an element of the relation R in the definition, and if X is a subset of an R-closed set (which is named E in the definition), then y has to be an element of E also.

Inductive sets

The formal definition of an inductive set defined by inference rules is based on the smallest set satisfying the relation. This set can be obtained by taking the intersection of all the R-closed sets of type T, i.e:

$$\bigcup\{E \mid E \text{ is an } R\text{-closed set}\}$$

In LaTTe, this translates as follows.

```
(definition inductive-set
  "The set inductively defined on `R`."
  [[T :type] [R (rules T)]]
  (intersections T (lambda [E (set T)]
                     (closed-set T R E))))
```

The definition of `intersections` (more precisely `latte-sets.powerset/intersections`) is as follows.

```
(intersections T X)
= (lambda [y T]
     (forall [x (set T)]
         (==> (set-elem T x X) (elem T y x))))
```

In the usual mathematical notation, this would be denoted by:

$$\{y : T \mid \forall x \in X,\ y \in x\}$$

Clearly there is a least fixed point hidden in the definition above, but in fact we do not need to make this explicit. One property that still remains important is that the inductive set is a lower bound for the set of R-closed sets.

```
(defthm inductive-set-lower-bound
  "If `Q` is an `R`-closed set, then the inductive
  set defined on `R` is included in `Q`."
  [[T :type] [R (rules T)] [Q (set T)]]
  (==> (closed-set T R Q)
       (subset T (inductive-set T R) Q)))
```

The proof follows.

```
(proof inductive-set-lower-bound
    :script
```

Our main assumption is that the set Q is R-closed.

```
  (assume [H (closed-set T R Q)]
```

The expected property directly results from the property that generalized intersections are lower bounds for the sets the range over.

```
    (have <a> (subset T (inductive-set T R) Q)
              :by ((pset/intersections-lower-bound T (lambda [E (set T)]
                                                       (closed-set T R E)))
                 Q H))
    (qed <a>)))
```

Another very important property is that the inductive sets (hence the generalized intersection) is the least R-closed set, i.e. it is itself an R-closed set.

```
(defthm closed-inductive-set
  "The set inductively defined on `R` is `R`-closed."
  [[T :type] [R (rules T)]]
  (closed-set T R (inductive-set T R)))
```

This means :

```
(forall [X (set T)]
      (forall [y T]
        (==> (R X y)
             (subset T X (inductive-set T R))
             (elem T y (inductive-set T R))))))
```

The proof is quite straightforward.

```
(proof closed-inductive-set
    :script
```

We have four assumptions: the assumption set X and the element y as well as the fact that they are in R. Moreover X is assumed to be a subset of the inductive set.

```
  (assume [X (set T)
           y T
           H1 (R X y)
           H2 (subset T X (inductive-set T R))]
```

Now consider an arbitrary set Y that is R-closed.

```
    (assume [Y (set T)
             HY (closed-set T R Y)]
```

We now that the inductive set is *below* Y in the subset relation

```
      (have a (subset T (inductive-set T R) Y)
            :by ((inductive-set-lower-bound T R Y) HY))
```

And by transitivity X is below Y (using hypothesis `H2`).

```
      (have b (subset T X Y)
            :by ((set/subset-trans T X (inductive-set T R) Y)
                 H2 a))
```

Hence y is an element of Y because the later is R-closed.

```
      (have c (elem T y Y) :by (HY X y H1 b))
```

And from this we reach the conclusion.

```
      (have d (forall [Y (set T)]
                  (==> (closed-set T R Y)
                         (elem T y Y)))
               :discharge [Y HY c]))
       (qed d)))
```

For convenience, we state the conditions for a term y to be an element of the inductive sets defined by rules R. (with the trivial proof).

```
(defthm elem-inductive-set
  "Membership for inductive set"
  [[T :type] [R (rules T)] [X (set T)] [y T]]
  (==> (R X y)
       (subset T X (inductive-set T R))
       (elem T y (inductive-set T R))))

(proof elem-inductive-set
   :script
   (assume [H1 (R X y)
            H2 (subset T X (inductive-set T R))]
      (have <a> (elem T y (inductive-set T R))
            :by ((closed-inductive-set T R) X y H1 H2))
      (qed <a>)))
```

Rule induction

The general principle of rule induction is then obtained as follows.

```
(defthm rule-induction
  "If a property `P` is `R`-closed, then each element of the
  inductive set verifies the property."
  [[T :type] [R (rules T)] [P (==> T :type)]]
  (==> (closed-set T R P)
       (forall [x T]
          (==> (elem T x (inductive-set T R))
               (P x)))))
```

The proof simply relies on the fact the the inductive-set is a lower bound.

```
(proof rule-induction
   :script
   (assume [H (closed-set T R P)]
       (have a (subset T (inductive-set T R) P)
             :by ((inductive-set-lower-bound T R P) H))
       (qed a)))
```

Examples

The set of natural numbers

We will now construct the set of natural number as an inductive set defined by rules.

```
(defaxiom nat
  ""
  []
  :type)

(defaxiom zero
  ""
  []
  nat)

(defaxiom succ
  ""
  []
  (==> nat nat))

(definition nat-rules
  "The inductive rules for the natural numbers."
  []
  (lambda [X (set nat)]
     (lambda [y nat]
        (or (and (seteq nat X (emptyset nat))
                 (equal nat y zero))
            (exists [n nat]
                (and (seteq nat X (lambda [k nat] (equal nat k n)))
                     (equal nat y (succ n))))))))

(definition nat-set
  "The inductive set of natural numbers."
  []
  (inductive-set nat nat-rules))

(defthm elem-seteq-equal
  "membership property of a singleton set"
  [[T :type] [s (set T)] [x T]]
  (==> (seteq T s (lambda [y T] (equal  T y x)))
       (elem T x s)))

(proof elem-seteq-equal
   :script
   (assume [H1 (seteq T s (lambda [y T] (equal T y x)))]
      (have <a> (elem T x (lambda [y T] (equal T y x)))
            :by (eq/eq-refl T x))
      (have <b> (elem T x s) :by ((p/%and-elim-right H1) x <a>))
      (qed <b>)))

(defthm nat-induct-closed
  "Rule induction for property `P` about
  natural numbers."
  [[P (==> nat :type)]]
  (==> (P zero)
       (forall [k nat] (==> (P k) (P (succ k))))
       (closed-set nat nat-rules P)))

(proof nat-induct-closed
    :script
  (assume [Hz (P zero)
           Hs (forall [k nat] (==> (P k) (P (succ k))))]
    (assume [N (set nat)
             n nat
             HNn (nat-rules N n)
             Hsub (subset nat N P)]
      "We proceed by case analysis on nat-rules"
      "The first case if if n=zero"
      (assume [Hzero (and (seteq nat N (emptyset nat))
                          (equal nat n zero))]
        (have <a1> (P n)
              :by ((eq/eq-subst nat P zero n)
                   ((eq/eq-sym nat n zero) (p/%and-elim-right Hzero))
                   Hz))
        (have <a> _ :discharge [Hzero <a1>]))
      "The second case if if n=(succ m) for some m."
      (assume [Hex (exists [m nat]
                     (and (seteq nat N (lambda [k nat] (equal nat k m)))
                          (equal nat n (succ m))))]
        (assume [m nat
                 Hm (and (seteq nat N (lambda [k nat] (equal nat k m)))
                         (equal nat n (succ m)))]
        (have <b1> (elem nat m N)
              :by ((elem-seteq-equal nat N m)
                   (p/%and-elim-left Hm)))
        (have <b2> (P m) :by (Hsub m <b1>))
        (have <b3> (P (succ m)) :by (Hs m <b2>))
        (have <b4> (P n) :by ((eq/eq-subst nat P (succ m) n)
                              ((eq/eq-sym nat n (succ m)) (p/%and-elim-right Hm))
                              <b3>))
        (have <b5> (forall [m nat]
                     (==> (and (seteq nat N (lambda [k nat] (equal nat k m)))
                               (equal nat n (succ m)))
                          (P n))) :discharge [m Hm <b4>]))
        (have <b6> (P n) :by ((q/ex-elim nat (lambda [m nat]
                                               (and (seteq nat N (lambda [k nat] (equal nat
                                                    (equal nat n (succ m)))) (P n))
                              Hex <b5>))
        (have <b> (==> (exists [m nat]
                         (and (seteq nat N (lambda [k nat] (equal nat k m)))
                              (equal nat n (succ m))))
                       (P n)) :discharge [Hex <b6>]))
      "Joining the two cases..."
      (have <c> (==> (==> (and (seteq nat N (emptyset nat))
                               (equal nat n zero))
                          (P n))
                     (==> (exists [m nat]
                            (and (seteq nat N (lambda [k nat] (equal nat k m)))
                                 (equal nat n (succ m))))
                          (P n))
                     (P n))
            :by ((p/or-elim (and (seteq nat N (emptyset nat))
                                 (equal nat n zero))
                            (exists [m nat]
                              (and (seteq nat N (lambda [k nat] (equal nat k m)))
                                   (equal nat n (succ m)))))
                 HNn (P n)))
      (have <d> (P n) :by (<c> <a> <b>))
      (have <e> _ :discharge [N n HNn Hsub <d>]))
    (qed <e>)))

(defthm nat-induction
  "Rule induction for property `P` about
  natural numbers."
  [[P (==> nat :type)]]
  (==> (P zero)
       (forall [k nat]
          (==> (P k) (P (succ k))))
       (forall-in [n nat nat-set] (P n))))

(proof nat-induction
   :script
   (assume [Hz (P zero)
            Hs (forall [k nat]
                  (==> (P k) (P (succ k))))]
      (have <a> (closed-set nat nat-rules P)
            :by ((nat-induct-closed P) Hz Hs))
      (have <b> (forall-in [n nat nat-set] (P n))
            :by ((rule-induction nat nat-rules P)
                 <a>))
      (qed <b>)))

(defthm zero-elem-nat
  "Zero is a natural number."
  []
  (elem nat zero nat-set))

(proof zero-elem-nat
    :script
  (have <a> (seteq nat (emptyset nat) (emptyset nat))
        :by (set/seteq-refl nat (emptyset nat)))
  (have <b> (equal nat zero zero) :by (eq/eq-refl nat zero))
  (have <c> _ :by (p/%and-intro <a> <b>))
  (have <d> (nat-rules (emptyset nat) zero)
        :by ((p/or-intro-left (and (seteq nat (emptyset nat) (emptyset nat)) (equal nat zero
                              (exists [n nat]
                                (and
                                 (seteq nat (emptyset nat) (lambda [k nat] (equal nat k n))
                                 (equal nat zero (succ n)))))
             <c>))
  (have <e> (subset nat (emptyset nat) nat-set)
        :by (set/emptyset-subset-lower-bound nat nat-set))
  (have <f> (elem nat zero nat-set)
        :by ((elem-inductive-set nat nat-rules (emptyset nat) zero)
             <d> <e>))
  (qed <f>))

(defthm succ-elem-nat
  "The successor of a natural number is a natural number."
  [[n nat]]
  (==> (elem nat n nat-set)
       (elem nat (succ n) nat-set)))

(proof succ-elem-nat
    :script
  (assume [H (elem nat n nat-set)]
    (have <a1> (seteq nat
                      (lambda [k nat] (equal nat k n))
                      (lambda [k nat] (equal nat k n)))
          :by (set/seteq-refl nat (lambda [k nat] (equal nat k n))))
    (have <a2> (equal nat (succ n) (succ n)) :by (eq/eq-refl nat (succ n)))
    (have <a3> (and (seteq nat
                           (lambda [k nat] (equal nat k n))
                           (lambda [k nat] (equal nat k n)))
                    (equal nat (succ n) (succ n)))
          :by (p/%and-intro <a1> <a2>))
    (have <a4> (exists [m nat]
                 (and (seteq nat
                             (lambda [k nat] (equal nat k n))
                             (lambda [k nat] (equal nat k m)))
                      (equal nat (succ n) (succ m))))
          :by ((q/ex-intro nat
                           (lambda [m nat]
                             (and (seteq nat
                                         (lambda [k nat] (equal nat k n))
                                         (lambda [k nat] (equal nat k m)))
                                  (equal nat (succ n) (succ m))))
                           n) <a3>))
    (have <a5> (nat-rules (lambda [k nat] (equal nat k n)) (succ n))
          :by ((p/or-intro-right
                (and (seteq nat
                            (lambda [k nat] (equal nat k n))
                            (emptyset nat))
                     (equal nat (succ n) zero))
                (exists [m nat]
                  (and (seteq nat
                              (lambda [k nat] (equal nat k n))
                              (lambda [k nat] (equal nat k m)))
                       (equal nat (succ n) (succ m)))))
               <a4>))
    (have <a> (nat-rules (lambda [k nat] (equal nat k n)) (succ n)) :by <a5>)
    (assume [p nat
             Hp (equal nat p n)]
      (have <b1> (elem nat p nat-set)
            :by ((eq/eq-subst nat nat-set n p)
                 ((eq/eq-sym nat p n) Hp)
                 H))
      (have <b> (subset nat
                        (lambda [k nat] (equal nat k n))
                        nat-set) :discharge [p Hp <b1>]))
    (have <c> (elem nat (succ n) nat-set)
          :by ((elem-inductive-set nat nat-rules
                                   (lambda [k nat] (equal nat k n))
```

```
                                    (succ n))
               <a> <b>))
    (qed <c>)))
```