lib/main/strong-induction.ath

```
## This file defines the strong-induction method, which transforms
## a strong-induction step,
##
## (forall ?n . (forall ?m . ?m < ?n ==> (P ?m)) ==> (P ?n))
##
## into an ordinary induction basis case (P zero) and induction step
## (forall ?n . (P ?n) ==> (P (S ?n))), from which (forall ?n . (P ?n))
## can be deduced using the built-in by-induction method.

load "nat-less"

module strong-induction {

define < := N.<

define (conclusion p n) := (urep (rename p) [n])

define (hypothesis p n) :=
  (forall ?m' (if (< ?m' n) (conclusion p ?m')))

define (step p) :=
  (forall ?n . (hypothesis p ?n) ==> (conclusion p ?n))

define (lemma p) := ((step p) ==> p)

define lemma-proof :=
  method (p)
    conclude (lemma p)
      assume (step p)
        let {sublemma :=
          by-induction (forall ?x . (hypothesis p ?x)) {
            zero =>
              conclude (hypothesis p zero)
                pick-any y:N
                  assume (y < zero)
                    (!from-complements (conclusion p y)
                     (y < zero) (!instance N.Less.not-zero [y]))
          | (S x) =>
             let {ind-hyp := (hypothesis p x)}
              conclude (hypothesis p (S x))
                pick-any y:N
                  assume (y < S x)
                    (!two-cases
                     assume (y = x)
                       (!chain-> [ind-hyp ==> (hypothesis p y) [(y = x)]
                                          ==> (conclusion p y) [(step p)]])
                     (!chain [(y =/= x)
                              ==> (y < S x & y =/= x)    [augment]
                              ==> (y < x)          [N.Less.S-step]
                              ==> (conclusion p y) [ind-hyp]]))
          }}
         conclude p
           pick-any x:N
             (!chain->
              [sublemma
               ==> (hypothesis p x) [(method (q) (!uspec q x))]
               ==> (conclusion p x) [(step p)]])

define principle :=
  method (p step-method)
    let {lemma := (!lemma-proof p);
         sp := conclude expected := (step p)
                 let {actual := pick-any n:N
                                  (!step-method n)}
                   (!sort-instance actual expected)}
      (!chain-> [sp ==> p [lemma]])
```

```


#===========================================================================
# This version accepts separate proofs of the basis case and induction step:

define principle2 :=
 method (p basis step-method)
  let {_ := datatype-cases (step p) {
              zero =>
              assume (hypothesis p zero)
                conclude (conclusion p zero)
                  (!basis)
            | (S n) =>
              let {_ := (!chain->
                         [true ==> ((S n) =/= zero) [N.S-not-zero]])}
                conclude ((hypothesis p (S n))
                          ==> (conclusion p (S n)))
                  (!step-method (S n))
            };
       lemma := (!lemma-proof p)}
    (!chain-> [(step p) ==> (forall ?n . (conclusion p ?n)) [lemma]])

#===========================================================================
# For testing strong induction step methods:

define (test-step step-method) :=
  letrec {p := (rename match (!step-method zero) {
                         (if ind-hyp body) => body
                       });
          goal := lambda (n)
                    (replace-term-in-prop zero p n);
          loop := lambda (n)
                    let {_ := (println (join "\nTesting " (val->string n) "..."));
                         _ := match (!step-method n) {
                                (if (forall (some-var x)
                                      (if (N.less x (val-of n)) (some-sent p)))
                                    (some-sent Q)) =>
                                let {p' := (goal x);
                                     Q' := (goal n);
                                     n-str := (val->string n)}
                                check {
                                  (&& (equal? p p') (equal? Q Q')) =>
                                  let {_ := (print
                                             (join "\nTest succeeded on "
                                                   n-str
                                                   ". Derived result:\n"
                                                   (val->string Q')))}
                                  (continue)
                                | else => (error
                                           (join "Test failed on " n-str))
                                }
                              }
                          }
                    (loop (S n))}
  (loop zero)

(define (measure-induction goal quantity conditional)
  (dmatch goal
  ((forall (some-var v) body)
    (dlet ((property (lambda (v') (replace-var v v' body)))
           (x (fresh-var))
           (IH (lambda (v)
                 (forall x (if (N.< (quantity x)
                                    (quantity v))
                               (property x)))))
           (goal-transformer (method (goal')
                                (dmatch goal'
                                  ((forall (some-var n) (forall (some-var x) ((if (= (some-term t) n) (some-sent concl
                                                        where (&& (equal? t (quantity x)) (equal? conclusion (property
                                    (pick-any y:(sort-of x)
```

```
                                       (conclude (property y)
                                         (dlet ((goal'' (!uspec goal' (quantity y)))
                                                (goal''' (!uspec goal'' y)))
                                           (!mp goal''' (!reflex (quantity y))))))))))
           ([n k] [(fresh-var "N") (fresh-var "N")])
           (Q (lambda (n)
                (forall x (if (= (quantity x) n)
                               (property x)))))
           (goal' (forall n (Q n)))
           (conditional (dmatch conditional
                           ((forall (some-var v') (some-sent body))
                              (dcheck ((equal? body (if (IH v') (property v')))
                                         (!claim conditional))))))
           (intermediate
             (!principle goal'
                (method (n)
                  (assume IND := (forall k (if (< k n) (Q k)))
                     conclude (Q n)
                        (pick-any x:(sort-of v)
                           (assume h1 := (= (quantity x) n)
                              (dlet ((conditional-x (!uspec conditional x))
                                     (IH-x (pick-any x'
                                             (assume h2 := (N.< (quantity x') (quantity x))
                                               (dlet ((S21 (conclude (if (< (quantity x')
                                                                            n)
                                                                         (Q (quantity x')))
                                                             (!uspec IND (quantity x'))))
                                                      (n=quantity-of-x (conclude (= n (quantity x))
                                                                         (!sym h1)))
                                                      (S22 (assume hyp := (< (quantity x') (quantity x))
                                                              (dlet ((hyp' (!chain-> [hyp ==> (< (quantity x') n) [h1]]
                                                                (!mp S21 hyp'))))
                                                      (S23 (!mp S22 h2))
                                                      (S24 (!uspec S23 x')))
                                                 (conclude (property x')
                                                   (!mp S24 (!reflex (quantity x')))))))))
                                      let {#_ := (print "\nconditional-x: " (val->string conditional-x) "\n");
                                           prop-x := (property x);
                                           conditional := (!sort-instance conditional-x (if IH-x prop-x))
                                          }
                                           ##(!chain-> [IH-x ==> (property x) [conditional-x]])
                                       (!mp conditional IH-x)
                                    ))))))))
     (!goal-transformer intermediate)))))


} # strong-induction
```