CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Section 8.1

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Natural Number Orderings

Goal: to reason about properties of natural number ordering relations, and its application to ordered lists and binary search trees.

- properties of natural number ordering functions
  - trichotomy properties
  - transitive and asymmetric properties
  - less-equal properties
  - combining ordering and arithmetic
- *natural number subtraction*
- *ordered lists*
- *binary search trees*

Natural number ordering functions

We begin with properties of a strict ordering operator $<$.

To define strict inequality on natural numbers, we introduce the following symbol declaration and module nested within module `N`.

Within module `N.Less` we define three axioms:

```
extend-module N {
  open Plus
  declare <: [N N] -> Boolean [[int->nat int->nat]]
  define [m n n' x y z k] := [?m:N ?n:N ?n':N ?x:N ?y:N ?z:N ?k:N]

  module Less {
    assert* def := [(zero < S n)
                    (~ _ < zero)
                    (S m < S n <==> m < n)]

    define [zero<S not-zero injective] := def
  } # close module Less
```

Natural number ordering functions

Within module `N.Less` we also define a number of sentences we intend to prove as theorems:

```
extend-module N {
  extend-module Less {

    define irreflexive  := (forall n . ~ n < n)
    define <S           := (forall n . n < S n)
    define =zero        := (forall n . ~ zero < n ==> n = zero)
    define zero<        := (forall n . n =/= zero <==> zero < n)
    define S1           := (forall x y . S x < y ==> x < y)
    define S2           := (forall x y . x < y ==> x < S y)
    define S4           := (forall m n . S m < n ==> exists n' . n = S n')
    define S-step       := (forall x y . x < S y & x =/= y ==> x < y)
    define discrete     := (forall n . ~ exists x . n < x & x < S n)

  } # close module Less
} # close module N
```

Natural number ordering functions

Each of these defined properties can be proved by induction.
The first two are quite easy (their proofs and those that follow are expressed within the scope of the module `N` extension, but outside the scope of module `N.Less`):

```
by-induction Less.irreflexive {
  zero =>  (!uspec Less.not-zero zero)
| (S n) => (!chain-> [(~ n < n) ==> (~ S n < S n) [Less.injective]])
}

by-induction Less.<S {
  zero =>  (!uspec Less.zero<S zero)
| (S n) => (!chain-> [(n < S n) ==> (S n < S S n) [Less.injective]])
}
```

Natural number ordering functions

More typically, proofs about < require case splitting in one or both of the basis case and inductive step, and proof by contradiction is also frequently useful, as in the following proof of `Less.S1`.

```
by-induction Less.S1 {
  zero =>
  conclude (forall y . S zero < y ==> zero < y)
    <D1>
| (S n) =>
  let {ind-hyp :=  (forall y . S n < y ==> n < y)}
    conclude (forall y . S S n < y ==> S n < y)
      <D2>
}
```

where `D1` and `D2` deductions prove the basis and inductive cases respectively.

Natural number ordering functions

The basis case D1 proving (forall y . S zero < y ==> zero < y) follows:

```
pick-any y
  assume Szero<y := (S zero < y)
    (!two-cases
      assume y=zero := (y = zero)
        <D1a>
      assume y!=zero := (y =/= zero)
        <D1b>
    )
```

where D1a and D1b deductions prove the $y = 0$ and $y \neq 0$ cases respectively.

Natural number ordering functions

The case `D1a` proving `(y = zero ==> zero < y)` follows:

```
(!by-contradiction (zero < y)
  assume (~ zero < y)
    let {-Szero<y :=
          conclude (~ S zero < y)
            (!chain->
              [true ==> (~ S zero < zero) [Less.not-zero]
                    ==> (~ S zero < y)    [y=zero]])}
      (!absurd Szero<y -Szero<y))
```

and `D1b` proving `(y =/= zero ==> zero < y)` follows:

```
let {has-predecessor :=
       (!chain-> [y!=zero
                  ==> (exists m . y = S m)    [nonzero-S]])}
  pick-witness m for has-predecessor
    (!chain-> [true ==> (zero < S m)      [Less.zero<S]
                    ==> (zero < y)        [(y = S m)]])
```

Natural number ordering functions

The inductive case `D2` proving `(forall y . S S n < y ==> S n < y)` follows:

```
pick-any y
  assume Less := (S S n < y)
    (!two-cases
      assume (y = zero)
        <D2a>
      assume nonzero := (y =/= zero)
        <D2b>
```

where `D2a` and `D2b` deductions prove the $y = 0$ and $y \neq 0$ cases respectively.

Natural number ordering functions

The case `D2a` proving `(y = zero ==> S n < y)` follows:

```
(!by-contradiction (S n < y)
  assume (~ S n < y)
    let {not-Less :=
          (!chain-> [true
                     ==> (~ S S n < zero)   [Less.not-zero]
                     ==> (~ S S n < y)      [(y = zero)]])}
      (!absurd Less not-Less))
```

Natural number ordering functions

The case D2b proving (y =/= zero ==> S n < y) follows:

```
              let {has-predecessor :=
                     (!chain->
                       [nonzero
                    ==> (exists m . y = S m)          [nonzero-S]])}
                  pick-witness m for has-predecessor
                   # we now have (y = S m)
                   (!chain->
                     [(S S n < y) ==> (S S n < S m)    [(y = S m)]
                                  ==> (S n < m)        [Less.injective]
                                  ==> (n < m)          [ind-hyp]
                                  ==> (S n < S m)      [Less.injective]
                                  ==> (S n < y)        [(y = S m)]]))
```

Note that in both cases D1b and D2b, we prove the existentially quantified sentence (exists m . y = S m) and pick a witness for it. We will see a simpler way of writing these parts of the proof.

Natural number ordering functions

Consider `Less.S2`:

```
(forall x y . x < y ==> x < S y)
```

The proof is much simpler:

```
by-induction Less.S2 {
  zero =>
    conclude (forall y . zero < y ==> zero < S y)
      pick-any y
        assume (zero < y)
          (!uspec Less.zero<S y)
| (S m) =>
    conclude (forall y . S m < y ==> S m < S y)
      pick-any y
        (!chain [(S m < y)
             ==> (m < y)                    [Less.S1]
             ==> (S m < S y)                [Less.injective]])
}
```

Natural number ordering functions

Notice the inductive hypothesis was not used. In such case, we can use Athena's `datatype-cases` form instead of `by-induction`.

```
datatype-cases Less.S2 {
  zero =>
    conclude (forall y . zero < y ==> zero < S y)
      pick-any y
        assume (zero < y)
          (!uspec Less.zero<S y)
| (S m) =>
    conclude (forall y . S m < y ==> S m < S y)
      pick-any y
        (!chain [(S m < y)
              ==> (m < y)                    [Less.S1]
              ==> (S m < S y)                [Less.injective]])
}
```

Other than the initial keyword, everything is the same as before.

Natural number ordering functions

Reconsidering `Less.S1`, we can use `datatype-cases` for the case splitting within both the basis case and the inductive step, thus significantly simplifying the proof:

```
by-induction Less.S1 {
  zero =>
    <D1'>
| (S n) =>
    let {ind-hyp := (forall y . S n < y ==> n < y)}
      <D2'>
} # by-induction Less.S1
```

where `D1'` and `D2'` deductions prove the basis and inductive cases respectively.

Natural number ordering functions

The basis case `D1'` proving

```
(forall y . S zero < y ==> zero < y)
```

follows:

```
datatype-cases (forall y . S zero < y ==> zero < y) {
  zero => assume less := (S zero < zero)
            let {-less := (!uspec Less.not-zero (S zero))}
              (!from-complements (zero < zero)
                                 less
                                 -less)
| (S m) => assume (S zero < S m)
             (!uspec Less.zero<S m)
}
```

Natural number ordering functions

The inductive case `D2'` proof follows:

```
      datatype-cases (forall y . S S n < y ==> S n < y) {
        zero =>
          assume less := (S S n < zero)
            let {-less := (!uspec Less.not-zero (S S n))}
              (!from-complements (S n < zero)
                                 less
                                 -less)
      | (S m) =>
          (!chain [(S S n < S m) ==> (S n < m)       [Less.injective]
                                 ==> (n < m)         [ind-hyp]
                                 ==> (S n < S m)     [Less.injective]])
      }
```

Trichotomy properties

- Strict inequality defines a *total ordering* on the natural numbers.
- The property expressing this is called *trichotomy*, reflecting the fact that one of three possibilities always holds.
- While it could be expressed as a disjunction of the three possibilities, other forms that are often more convenient in proofs are stated here:

```
extend-module Less {
  define trichotomy  := (forall m n . ~ m < n & m =/= n ==> n < m)
  define trichotomy1 := (forall m n . ~ m < n & ~ n < m ==> m = n)
  define trichotomy2 := (forall m n . m = n <==> ~ m < n & ~ n < m)
}
```

Trichotomy properties

The basis case of the proof of trichotomy by induction follows:

```
by-induction Less.trichotomy {
  zero =>
    pick-any n
      assume (~ zero < n & zero =/= n)
        conclude (n < zero)
          let {has-pred := (!chain-> [(zero =/= n)
                                      ==> (n =/= zero)          [sym]
                                      ==> (exists k . n = S k) [nonzero-S]])}
            pick-witness k for has-pred
              let {less := (!chain-> [true ==> (zero < S k) [Less.zero<S]
                                           ==> (zero < n) [(n = S k)]]);
                   -less := (~ zero < n)}
                (!from-complements (n < zero) less -less)
```

Trichotomy properties

The inductive step follows:

```
| (S m) =>
    let {ind-hyp := (forall n . ~ m < n & m =/= n ==> n < m)}
      datatype-cases (forall n . ~ S m < n & S m =/= n ==> n < S m) {
        zero => assume (~ S m < zero & S m =/= zero)
                  (!uspec Less.zero<S m)
      | (S k) => assume A := (~ S m < S k & S m =/= S k)
                   (!chain->
                      [A ==> (~ m < k & m =/= k) [Less.injective
                                                  S-injective]
                         ==> (k < m)             [ind-hyp]
                         ==> (S k < S m)         [Less.injective]])
      }
}
```

Transitive and asymmetric properties

The next collection of properties of < concerns transitivity:

```
extend-module Less {
  define transitive   := (forall x y z . x < y & y < z ==> x < z)
  define transitive1  := (forall x y z . x < y & ~ z < y ==> x < z)
  define transitive2  := (forall x y z . x < y & ~ x < z ==> z < y)
  define transitive3  := (forall x y z . ~ y < x & y < z ==> x < z)
}
```

Transitive and asymmetric properties

Proving transitivity by induction requires case splitting on zero and nonzero values for both y and z.

Instead, we follow the approach of commuting universally quantified variables, and then use the `chain` method to deduce the original property as follows:

```
conclude Less.transitive
 let {transitive0 :=
          # A version with the easiest-to-induct-on variable first:
            (forall z x y . x < y & y < z ==> x < z);
          <D>
     }
     pick-any x y z
       (!chain [(x < y & y < z) ==> (x < z) [transitive0]])
```

where `D` corresponds to the deduction of the alternative property by induction.

Transitive and asymmetric properties

The deduction `D` by induction on `z` for `(forall z x y . x < y & y < z ==> x < z)` follows (basis case):

```
    _ := by-induction transitive0 {
           zero =>
             pick-any x y
               assume (x < y & y < zero)
                 let {-y<0 := (!uspec Less.not-zero y)}
                   (!from-complements (x < zero) (y < zero) -y<0)
         | (S n) =>
           <D'>
         } # close by-induction
```

Transitive and asymmetric properties

The deduction `D'` for the inductive case `z = S n`, i.e., `(forall x y . x < y & y < S n ==> x < S n)` follows:

```
let {ind-hyp := (forall x y . x < y & y < n ==> x < n)}
 pick-any x y
  assume (x < y & y < S n)
   conclude (x < S n)
    let {_ := conclude (x < n)
                (!two-cases
                  assume (y = n)
                    (!chain-> [(x < y) ==> (x < n) [(y = n)]])
                  assume (y =/= n)
                    (!chain-> [(y =/= n)
                       ==> (y < S n & y =/= n) [augment]
                       ==> (y < n)             [Less.S-step]
                       ==> (x < y & y < n)     [augment]
                       ==> (x < n)             [ind-hyp]]))}
          (!chain-> [(x < n) ==> (x < S n)       [Less.S2]])
```

Transitive and asymmetric properties

Another key property of < is *asymmetry*:

```
extend-module Less {
  define asymmetric   := (forall m n . m < n ==> ~ n < m)
}
```

With the transitive and irreflexive properties at hand to use as lemmas, `Less.asymmetric` doesn't even require induction:

```
conclude Less.asymmetric
  pick-any x y
    assume (x < y)
      (!by-contradiction (~ y < x)
         assume (y < x)
           let {x<x := (!chain-> [(x < y & y < x)
                                  ==> (x < x)        [Less.transitive]]);
                -x<x := (!uspec Less.irreflexive x)}
             (!absurd x<x -x<x))
```

Transitive and asymmetric properties

The following property is an easy consequence of `Less.S1` and irreflexivity:

```
extend-module Less {
  define S-not-< := (forall n . ~ S n < n)
}

conclude Less.S-not-<
  pick-any n
    (!by-contradiction (~ S n < n)
      assume (S n < n)
        (!absurd
          (!chain-> [(S n < n) ==> (n < n)  [Less.S1]])
          (!uspec Less.irreflexive n)))
```

Less-equal properties

Other ordering operators—$\leq$, $>$, and $\geq$—can be defined in terms of $<$ and $=$. We restrict our attention here to $\leq$:

```
declare <=: [N N] -> Boolean [[int->nat int->nat]]

module Less= {

  assert definition := (forall x y . x <= y <==> x < y | x = y)

  define Implied-by-<     := (forall m n . m < n ==> m <= n)
  define Implied-by-equal := (forall m n . m = n ==> m <= n)
  define reflexive        := (forall n . n <= n)
  define zero<=           := (forall n . zero <= n)
  define S-zero-S-n       := (forall n . S zero <= S n)
  define injective        := (forall n m . S n <= S m <==> n <= m)
  define not-S            := (forall n . ~ S n <= n)
  define S-not-equal      := (forall k n . S k <= n ==> k =/= n)
  define discrete         := (forall m n . m < n ==> S m <= n)
}
```

Less-equal properties

More properties defined for $\leq$:

```
extend-module Less= {
  define transitive  := (forall x y z . x <= y & y <= z ==> x <= z)
  define transitive1 := (forall x y z . x < y & y <= z ==> x < z)
  define transitive2 := (forall x y z . x <= y & y < z ==> x < z)
  define S1          := (forall n m . n <= m ==> n < S m)
  define S2          := (forall n m . n <= m ==> n <= S m)
  define S3          := (forall n . n <= S n)
  define trichotomy1 := (forall m n . ~ n <= m ==> m < n)
  define trichotomy2 := (forall m n . ~ n < m ==>  m <= n)
  define trichotomy3 := (forall m n . n < m ==> ~ m <= n)
  define trichotomy4 := (forall m n . n <= m ==> ~ m < n)
  define trichotomy5 := (forall m n . m <= n & n <= m ==> m = n)
}
```

Less-equal properties

And even more properties defined for $\leq$:

```
extend-module Less= {
  define Plus-cancellation := (forall k m n . m + k <= n + k ==> m <= n)
  define Plus-k       := (forall k m n . m <= n ==> m + k <= n + k)
  define Plus-k1      := (forall k m n . m <= n ==> m <= n + k)
  define k-Less=      := (forall k m n . n = m + k ==> m <= n)
  define zero2        := (forall n . n <= zero ==> n = zero)
  define not-S-zero   := (forall n . ~ S n <= zero)
  define S4           := (forall m n . S m <= n ==> exists n' . n = S n')
  define S5           := (forall n m . n <= S m & n =/= S m ==> n <= m)
  define =zero        := (forall m . m < one ==> m = zero)
  define zero<=one    := (forall m . m = zero ==> m <= one)
}
```

Only `Less=.definition` needs to be asserted as an axiom; the rest are theorems provable from `Less=.definition` and properties of < and equality. (Induction is not necessary.)

Less-equal properties

The first two properties follow simply from the definition, but offer examples of using `alternate` in implication chains.

```
conclude Less=.Implied-by-<
  pick-any m n
    (!chain [(m < n) ==> (m < n | m = n)   [alternate]
                     ==> (m <= n)          [Less=.definition]])

conclude Less=.Implied-by-equal
  pick-any m:N n:N
    (!chain [(m = n) ==> (m < n | m = n)   [alternate]
                     ==> (m <= n)          [Less=.definition]])

conclude Less=.reflexive
  pick-any n
    (!chain-> [(n = n) ==> (n <= n)        [Less=.Implied-by-equal]])
```

Less-equal properties

To prove `Less=.zero<=`, we split into zero and nonzero cases, for which the best tool at hand is `datatype-cases`:

```
datatype-cases Less=.zero<= {
  zero =>  (!uspec Less=.reflexive zero)        # zero <= zero
| (S n) =>
   (!chain-> [true ==> (zero < S n)             [Less.zero<S]
                   ==> (zero <= S n)            [Less=.Implied-by-<]])
}
```

Less-equal properties

The proof of `Less=.injective` provides a nice illustration of the power of equivalence chaining:

```
conclude Less=.injective
  pick-any n m
    (!chain [(S n <= S m)
        <==> (S n < S m | S n = S m)  [Less=.definition]
        <==> (n < m | n = m)          [Less.injective S-injective]
        <==> (n <= m)                 [Less=.definition]])
```

Less-equal properties

Since `Less=.not-S` is a (quantified) negation, we use proof by contradiction:

```
conclude Less=.not-S
  pick-any n
    (!by-contradiction (~ S n <= n)
      assume Sn<=n := (S n <= n)
        let {disjunction :=
               (!chain-> [Sn<=n ==> (S n < n | S n = n) [Less=.definition]])}
          (!cases disjunction
            assume Sn<n := (S n < n)
              let {-Sn<n := (!chain-> [true ==> (~ Sn<n) [Less.S-not-<]])}
                (!absurd Sn<n -Sn<n)
            assume Sn=n := (S n = n)
              let {-Sn=n := (!chain-> [true ==> (~ Sn=n) [S-not-same]])}
                (!absurd Sn=n -Sn=n)))
```

Less-equal properties

The following proof of `Less=.discrete` provides an interesting example of generating an existential property in order to contradict the negated one in `Less.discrete`:

```
conclude Less=.discrete
  pick-any m n
    assume (m < n)
      (!by-contradiction (S m <= n)
        assume -Sm<=n := (~ S m <= n)
          let {in-between := (exists k . m < k & k < S m)}
            (!absurd
              (!chain-> [-Sm<=n
                         ==> (~ (S m < n | S m = n))   [Less=.definition]
                         ==> (~ S m < n & S m =/= n)   [dm]
                         ==> (n < S m)                 [Less.trichotomy]
                         ==> (m < n & n < S m)         [augment]
                         ==> in-between                [existence]])
              (!uspec Less.discrete m)))  # ~ in-between
```

Less-equal properties

A proof of `Less=.zero2` is possible using `dsyl` (disjunctive syllogism) and the proof of `Less=.not-S-zero` is a proof by contradiction:

```
conclude Less=.zero2
  pick-any n
    assume hyp := (n <= zero)
      (!dsyl
         (!chain-> [hyp ==> (n < zero | n = zero) [Less=.definition]])
         (!uspec Less.not-zero n))  # (~ n < zero)

conclude Less=.not-S-zero
  pick-any n
    (!by-contradiction (~ S n <= zero)
      assume hyp := (S n <= zero)
        (!absurd
          (!chain-> [hyp  ==> (S n = zero)   [Less=.zero2]])
          (!uspec S-not-zero n)))  # (S n =/= zero)
```

Combining ordering and arithmetic

Consider interaction between ordering operators and addition:

```
extend-module Less {
  define Plus-cancellation := (forall k m n . m + k < n + k ==> m < n)
  define Plus-k := (forall k m n . m < n ==> m + k < n + k)
}
by-induction Less.Plus-cancellation {
  zero =>
    pick-any m n
      (!chain [(m + zero < n + zero)
           ==> (m < n)                    [Plus.right-zero]])
| (S k) =>
    let {induction-hypothesis := (forall m n . m + k < n + k ==> m < n)}
      pick-any m n
        (!chain [(m + S k < n + S k)
             ==> (S (m + k) < S (n + k)) [Plus.right-nonzero]
             ==> (m + k < n + k)         [Less.injective]
             ==> (m < n)                 [induction-hypothesis]])
}
```

Combining ordering and arithmetic

For the proof of `Less.Plus-k` we use `Less=.Plus-cancellation` as well as a couple of the trichotomy properties.

```
conclude Less.Plus-k
  pick-any k m n
    assume hyp1 := (m < n)
      let {goal := (m + k < n + k)}
        (!by-contradiction goal
         (!chain [(~ goal)
                  ==> (n + k <= m + k)   [Less=.trichotomy2]
                  ==> (n <= m)           [Less=.Plus-cancellation]
                  ==> (~ m < n)          [Less=.trichotomy4]
                  ==> (hyp1 & ~ m < n)   [augment]
                  ==> false              [prop-taut]]))
```

Similar properties hold between ordering relations and natural number multiplication. See the Athena library file `lib/main/nat-less.ath`.