15-414: Bug Catching: Automated Program Verification

Lecture Notes on Induction

Frank Pfenning

Carnegie Mellon University

Lecture 11
Thursday, March 11, 2021

1 Introduction

After analyzing first-order logic in the last lecture, we broach the subject of arithmetic and specifically induction in this lecture. In particular, we'll explore ways to carry out induction in Why3.

After a few introductory examples we explore the fundamentally inductive nature of α^* in dynamic logic—something we have avoided so far. This will allow us to prove the induction axiom for reasoning about programs, which we were unable to do before.

Learning goals. After this lecture, you should be able to:

- Use explicit induction in Why3;
- Reason about the semantic validity of axioms in dynamic logic using mathematical induction.

2 Peano's Axioms

The theory of arithmetic as axiomatized by Peano concerns the natural numbers $0, 1, 2, \ldots$ while in Why3 we have been dealing with integers $\ldots, -2, -1, 0, 1, 2, \ldots$. This gap is not difficult to bridge, but in this lecture we will focus on natural numbers. You can read more about Peano's Axioms on Wikipedia.

He starts with axioms on equality, such as reflexivity, symmetry, and transitivity of equality. Furthermore, there is a constant 0 and successor function S such that $\mathsf{S}(n) =$

$\mathsf{S}(k)$ iff $n = k$. Furthermore, $0 \neq \mathsf{S}(n)$ for all n. Finally, we have the induction axiom

$$P(0) \wedge (\forall n.\, P(n) \to P(\mathsf{S}(n)) \to \forall n. P(n)$$

where $P(n)$ is an arbitrary predicate on natural numbers. In the context of this course, $P(n)$ is a property of n definable by a formula.

We can also write the axiom as an inference rule in the sequent calculus.

$$\frac{\Gamma \vdash P(0), \Delta \quad \Gamma, P(n) \vdash P(\mathsf{S}(n)), \Delta}{\Gamma \vdash P(e), \Delta}\,\mathsf{ind}$$

where n does not already occur in Γ, $P(e)$, or Δ. This rule is formulated in this particular manner because we like to avoid mentioning other connectives are quantifiers and keep the rules “pure”.

3 Induction in Why3

Among the Why3 standard libraries we find `int.SimpleInduction` for mathematical induction and also `int.Induction` for complete induction.

```
module SimpleInduction

  use Int

  predicate p int

  axiom base: p 0

  axiom induction_step: forall n:int. 0 <= n -> p n -> p (n+1)

  lemma SimpleInduction : forall n:int. 0 <= n -> p n

end
```

This module keeps the predicate p as well as the base and induction_step axioms *abstract* and provides the lemma SimpleInduction. The idea is to *clone* this module by providing a concrete predicate for p. Secondly, we want to turn base and induction_step into *lemmas*, which creates corresponding proof obligations. If they can be proved, we obtain those two lemmas, plus SimpleInduction as a consequence.

Here is an example. We would like to prove

$$\sum_{i=0}^{i=n} i = \frac{n(n+1)}{2}$$

We start by defining a sum function with a lower bound of a and upper bound of b, inclusive.

```
  let rec function sum (a : int) (b : int) : int =
  variant { b - a }
  if a > b then 0 else sum a (b-1) + b
```

Then we define the predicate $P(n)$ intended for the induction, here called sum_square.

```
predicate sum_square (n : int) =
  n >= 0 -> sum 0 n = div (n*(n+1)) 2
```

We can now clone the standard library as sketched above.

```
clone int.SimpleInduction
  with predicate p = sum_square, lemma base, lemma induction_step
```

The summary of the whole theory is below. Just to be sure, we also restate the desired property as a *goal*. This generates a proof obligation just like a lemma, but does not assume the proven formula. This is helpful if we'd like to avoid polluting the search space.

```
theory SumSquare1

  use int.Int
  use int.EuclideanDivision

  let rec function sum (a : int) (b : int) : int =
  variant { b - a }
  if a > b then 0 else sum a (b-1) + b

  predicate sum_square (n : int) =
    n >= 0 -> sum 0 n = div (n*(n+1)) 2

  clone int.SimpleInduction
    with predicate p = sum_square, lemma base, lemma induction_step

  goal G : forall n:int. n >= 0 -> sum_square n

end
```

4 Induction via Recursion

Besides the technique above we can also often provide an explicit witness as a function. When we can prove the termination of such a witness function, and the contract expresses the desired property, then the theorem we would like to state becomes an immediate consequence.

Here is a first way of achieving this[1].

```
theory SumSquare3

  use int.Int
  use int.EuclideanDivision

  let rec function sum (a : int) (b : int) : int =
  variant { b - a }
  if a > b then 0 else sum a (b-1) + b

```

[1]which we did not do in lecture

```
  let rec function sum_rec (n : int) =
  requires { n >= 0 }
  variant { n }
  ensures { result = div (n * (n+1)) 2 }
  ensures { result = sum 0 n }
  if n = 0 then 0
  else sum_rec (n-1) + n

  goal G : forall n:int. n >= 0 -> sum 0 n = div (n * (n+1)) 2

end
```

The witness function here is sum_rec, with an explicit property statement following. Essentially, sum_rec is a *circular proof* of the desired property, acceptable because the variant shows that it is terminating.

An even slicker formulation of this technique[2] is for the witness function to return a Boolean, in effect making it a lemma. The key here is that the postcondition must guarantee the property we ultimately want, so it might use the same predicate we used with the explicit induction axiom.

```
theory SumSquare2

  use int.Int
  use int.EuclideanDivision

  let rec function sum (a : int) (b : int) : int =
  variant { b - a }
  if a > b then 0 else sum a (b-1) + b

  predicate sum_square (n : int) =
  n >= 0 -> sum 0 n = div (n * (n+1)) 2

  let rec lemma sq (n : int) =
  requires { n >= 0 }
  variant { n }
  ensures { result /\ sum_square n }
  if n = 0 then true
  else sq (n-1)

  goal G : forall n:int. n >= 0 -> sum_square n

end
```

The verification condition in the $n = 0$ branch is exactly the base of the induction, and the condition in the $n > 0$ branch is the induction step. In the tail call sq $(n - 1)$ we have to satisfy the precondition $(n \geq 0)$ which is true, and then get to assume the postcondition on the actual parameter, that is, sum_square $(n - 1)$. We have to prove sum_square n, precisely the implication that represents the induction step.

[2]which we *did* briefly show in lecture

5 Revisiting Dynamic Logic in Why3

When we gave the mathematical definition of repetition in Dynamic logic we wrote

$$\omega_0[\![\alpha^*]\!]\omega_n \quad \text{iff there exist } \omega_1, \ldots, \omega_{n-1} \text{ such that } \omega_i[\![\alpha]\!]\omega_{i+1} \text{ for all } 0 \leq i < n.$$

In our formalization we instead used the axiom

```
axiom run_star : forall omega alpha nu.
  run omega (Star alpha) nu
  <-> omega = nu \/ exists mu. run omega alpha mu /\ run mu (Star
      alpha) nu
```

This formalization is considerably weaker since it does not allow us to prove properties of α^* by induction. In particular, we are unable to prove the dynamic logic axiom of induction:

```
 axiom induction : forall omega alpha q.
   models omega (Box (Star alpha) q)
   <-> models omega (And q (Box (Star alpha)
                             (Implies q (Box alpha q))))
```

Our mathematical argument for this property relied on induction on n as used in the mathematical definition of $\omega_0[\![\alpha^*]\!]\omega_n$.

Now that we have discovered how to carry out inductive arguments in Why3 we can revised our definition to match the mathematical definition more closely and then use that to actually prove the induction axiom. The complete code can be found in ndl.mlw.

As a first step, we need two predicates: run as before and run_bdd that a relation running a program α for n times in succession, written as $\omega[\![\alpha]\!]^n\nu$. Mathematically we define

$$\begin{array}{ll} \omega[\![\alpha^*]\!]\nu & \text{iff there exists an } n \text{ such that } \omega[\![\alpha]\!]^n\nu \\ \omega[\![\alpha]\!]^0\nu & \text{iff } \omega = \nu \\ \omega[\![\alpha]\!]^{n+1}\nu & \text{iff there exists a } \mu \text{ such that } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\alpha]\!]^n\mu \end{array}$$

Our formalization in Why3 models the mathematical definition very closely.

```
predicate run (omega : state) (alpha : prog) (nu : state)
predicate run_bdd (omega : state) (alpha : prog) (n : int) (nu :
    state)

axiom run_star : forall omega alpha nu.
  run omega (Star alpha) nu
  <-> exists n. n >= 0 /\ run_bdd omega alpha n nu
axiom run_bdd : forall omega alpha nu n.
   run_bdd omega alpha n nu
   <-> (n = 0 /\ omega = nu)
     \/ (n > 0 /\ exists mu. run omega alpha mu
                            /\ run_bdd mu alpha (n-1) nu)
```

We elide the remaining axioms for assignment, sequential composition, nondeterministic choice, and guards because they have not changed from Lecture 8.

Why3 can now prove the original axiom, which corresponds to unfolding the repetition (implication from left to right) and folding the repetition (from right to left). Now induction is required here.

```
lemma run_star_fold : forall omega alpha nu.
  (omega = nu \/ exists mu. run omega alpha mu
                       /\ run mu (Star alpha) nu)
  -> run omega (Star alpha) nu

lemma run_star_unfold : forall omega alpha nu.
  run omega (Star alpha) nu
  -> (omega = nu \/ exists mu. run omega alpha mu
                          /\ run mu (Star alpha) nu)
```

Similarly, the loop unfolding axiom of dynamic logic, namely

$$[\alpha^*]Q \leftrightarrow Q \wedge [\alpha][\alpha^*]Q$$

is easy to prove in both directions.

```
lemma models_box_star : forall omega alpha q.
  models omega (Box (Star alpha) q)
  <-> models omega (And q (Box alpha (Box (Star alpha) q)))
```

As before, we have mapped the bi-implication from dynamic logic directly to bi-implication in Why3.

What remains now is to prove the representation of the induction axiom in Why3, which was impossible before.

6 Proving Induction for Dynamic Logic in Why3

We would like to prove the validity of computational induction in dynamic logic:

$$[\alpha^*]Q \leftrightarrow Q \wedge [\alpha^*](Q \rightarrow [\alpha]Q)$$

In our representation both directions of this bi-implication require induction.

We start by proving the left-to-right direction using several lemmas. We first state them in ordinary mathematical language

For all $n \geq 0$, ω, and ν,
if $\omega[\![\alpha]\!]^n\mu$ and $\mu[\![\alpha]\!]\nu$ for some μ
then $\omega[\![\alpha]\!]\mu'$ and $\mu'[\![\alpha]\!]^n\nu$ for some μ'

We prove this by induction on n. In Why3:

```
predicate run_bdd_left (n:int) =
  forall omega alpha nu.
    (exists mu. run_bdd omega alpha n mu /\ run mu alpha nu)
    -> (exists mu'. run omega alpha mu' /\ run_bdd mu' alpha n nu)

clone int.SimpleInduction as Bdd1
  with predicate p = run_bdd_left, lemma base, lemma induction_step
```

Note that we clone `int.SimpleInduction` with an explicit name `Bdd1` so that we can clone the same module later without creating a name conflict. We can similarly prove the opposite direction of this implication.

```
predicate run_bdd_right (n:int) =
  forall omega alpha nu.
    (exists mu'. run omega alpha mu' /\ run_bdd mu' alpha n nu)
    -> (exists mu. run_bdd omega alpha n mu /\ run mu alpha nu)

clone int.SimpleInduction as Bdd2
  with predicate p = run_bdd_right, lemma base, lemma
      induction_step
```

With these, we can prove an alternative characterization of n-fold repetition, namely a “right-associative” composition instead of the original “left-associative” one.

$$\begin{array}{ll} \omega[\![\alpha]\!]^0\nu & \text{iff } \omega = \nu \\ \omega[\![\alpha]\!]^{n+1}\nu & \text{iff there exists a } \mu \text{ such that } \omega[\![\alpha]\!]^n\mu \text{ and } \mu[\![\alpha]\!]\nu \end{array}$$

This can be proved directly from the two preceding lemmas.

```
lemma run_bdd_alt : forall omega alpha nu n.
  run_bdd omega alpha n nu
  <-> (n = 0 /\ omega = nu)
   \/ (n > 0 /\ exists mu. run_bdd omega alpha (n-1) mu /\ run mu
      alpha nu)

lemma box_star_0 : forall omega alpha q.
  models omega (Box (Star alpha) q) -> models omega q

lemma box_star_left : forall omega alpha q.
  models omega (Box (Star alpha) q)
  -> models omega (Box (Star alpha) (Box alpha q))

lemma induction_unwind : forall omega alpha q.
  models omega (Box (Star alpha) q)
  -> models omega (And q (Box (Star alpha)
                              (Implies q (Box alpha q))))
```

The other direction of the induction axiom is intuitively more complicated, but follows here just by one lemma, proved with a simple induction.

```
predicate run_bdd_inv (n : int) =
  forall omega alpha q.
    models omega q /\ models omega (Box (Star alpha)
                                        (Implies q (Box alpha q)))
    -> forall nu. run_bdd omega alpha n nu -> models nu q

clone int.SimpleInduction as Bdd3
  with predicate p = run_bdd_inv, lemma base, lemma induction_step

lemma induction_wind : forall omega alpha q.
  models omega (And q (Box (Star alpha) (Implies q (Box alpha q))))
  -> models omega (Box (Star alpha) q)
```

7 Induction with Invariants

Induction with invariants just requires a few lemmas regarding validity (as expressed with $\Box P$) and the properties we have already proved. We just summarize them here, since they introduce no new and interesting ideas.

```
lemma valid_box : forall omega alpha p.
  models omega (Valid p) -> models omega (Box alpha p)

lemma dist_box_implies : forall omega alpha p q.
  models omega (Box alpha (Implies p q))
  -> models omega (Box alpha p)
  -> models omega (Box alpha q)

lemma dist_valid_box : forall omega alpha p q.
  models omega (Valid (Implies p q))
  -> models omega (Box alpha p)
  -> models omega (Box alpha q)

lemma induction_valid : forall omega q alpha.
  models omega (And q (Valid (Implies q (Box alpha q))))
  -> models omega (Box (Star alpha) q)

lemma induction_inv : forall omega j alpha q.
  models omega (And j
                  (And (Valid (Implies j (Box alpha j)))
                       (Valid (Implies j q))))
  -> models omega (Box (Star alpha) q)
```

The final theorem here is what we were aiming at, namely

$$J \wedge \Box(J \rightarrow [\alpha]J) \wedge \Box(J \rightarrow Q) \rightarrow [\alpha^*]Q$$

What we did not accomplish and still remains to be done is the formalization of the axiom of *convergence* which concerns $\langle\alpha^*\rangle Q$.