15-414: Bug Catching: Automated Program Verification

Lecture Notes on Induction

Matt Fredrikson

Carnegie Mellon University
Lecture 10
September 25, 2025

1 Introduction

In Lecture 7 we analyzed reasoning about while loops. To do so, we introduced the more abstract nondeterministic dynamic logic (NDL) with nondeterministic choice and iteration. Eventually, we arrived at two axioms for iteration in NDL: unrolling and induction:

$$[\alpha^*]Q \leftrightarrow Q \wedge [\alpha][\alpha^*]Q \qquad \text{unrolling}$$
$$[\alpha^*]Q \leftrightarrow Q \wedge [\alpha^*](Q \rightarrow [\alpha]Q) \qquad \text{induction}$$

From these and the other axioms, we were able to derive a *practical* version of the axiom for reasoning about while loops with loop invariants J. This is no longer an "if and only if" but nevertheless allows us to prove properties of while loops.

$$[\mathsf{while}\, P\, \alpha]Q \leftarrow J \wedge \Box(J \wedge P \rightarrow [\alpha]J) \wedge \Box(J \wedge \neg P \rightarrow Q)$$

Recall that $\Box P$ is true if P is valid, which means true in every possible state.

This final axiom is precisely the form that Why3 (and other systems) use to generate verification conditions for loops when provided with a loop invariant J. We still reason about *partial correctness*, that is, this axiom only ensures that *if the program terminates* then the postcondition Q will hold in the final state. In the next lecture we will address *total correctness*, which also requires termination.

Our formalization of NDL in Why3 in Lecture 8 proved correctness for various axioms allowing decomposition of α in $[\alpha]Q$, but omitted NDL axioms for assignment and iteration. The first one is possible, but tedious due to the side condition on occurrences of variables; the second one is the subject of this lecture. So our goal is to prove the correctness of the induction axiom in NDL.

Perhaps not surprisingly, this seems to require proof by *induction*. In general proofs by induction are difficult to automate, so they are not natively available in Why3 or the back-end provers such as alt-ergo, Z3, or CVC4. Instead we have to "trick" Why3 into carrying out such proofs for us. We show how to do this, and then apply the learned techniques to the induction axiom in NDL.

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

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

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.

3 Induction Proofs in NDL

So far, we have been using NDL and its axioms to turn reasoning about programs into problems of logical reasoning. Now we turn the tables: we want to see how to model logical reasoning by program reasoning! The goal, in short, is to write a program α so that $[\alpha]P(i)$ is true if there is an induction proof $\forall i.\, P(i)$. Actually, we want something slightly stricter, namely to encoding particular induction schemas. In the first case, it will simply Peano's axiom.

So consider the following proposition in NDL:

$$[i \leftarrow 0 \,;\, (i \leftarrow i + 1)^*]P(i)$$

Because the loop may be executed any number of times $0, 1, 2, \ldots$, this proposition is valid in NDL if $P(i)$ holds for all $i \geq 0$. Let's reason through a verification effort using

our axioms for NDL:

$$\begin{array}{ll} & \models [i \leftarrow 0 \,;\, (i \leftarrow i+1)^*]P(i) \\ \text{iff} & \models [i \leftarrow 0][(i \leftarrow i+1)^*]P(i) \\ \text{iff} & \models \forall i.\, i = 0 \rightarrow [(i \leftarrow i+1)^*]P(i) \\ \text{if} & \models \forall i.\, i = 0 \rightarrow P(i) \wedge \Box(P(i) \rightarrow [i \leftarrow i+1]P(i)) \end{array}$$

Here, the last step used the form of the induction axiom where we have replaced $[\alpha^*]Q$ by $\Box Q$. We can take some further steps to eliminate the $\Box$ modality.

$$\begin{array}{ll} & \models \forall i.\, i = 0 \rightarrow P(i) \wedge \Box(P(i) \rightarrow [i \leftarrow i+1]P(i)) \\ \text{iff} & \models \forall i.\, i = 0 \rightarrow P(i) \wedge \Box(P(i) \rightarrow \forall i'.\, i' = i+1 \rightarrow P(i')) \\ \text{iff} & \models (\forall i.\, i = 0 \rightarrow P(i)) \wedge \forall i.\, i = 0 \rightarrow \Box(P(i) \rightarrow \forall i'.\, i' = i+1 \rightarrow P(i')) \\ \text{iff} & \models P(0) \wedge \forall i.\, i = 0 \rightarrow \Box(P(i) \rightarrow P(i+1)) \end{array}$$

In the second conjunct, we lose the assumption about $i = 0$ underneath the $\Box$ modality because we have to prove that $P(i) \rightarrow P(i+1)$ regardless of the state. We can formulate the upshot of this line of reasoning as

$$\models [i \leftarrow 0 \,;\, (i \leftarrow i+1)^*]P(i) \text{ if } \models P(0) \wedge \forall i.\, P(i) \rightarrow P(i+1)$$

In other words, using the induction axiom in NDL, we can reduce the proof of proving the validity of $[i \leftarrow 0 \,;\, (i \leftarrow i+1)^*]P(i)$ to proving the validity of $P(0) \wedge \forall i.\, P(i) \rightarrow P(i+1)$, which should have been intuitively clear when we started.

4 Induction in Why3

Unfortunately, Why3 doesn't allow us to write a nondeterministic program as from the previous section, or verify it in a "disembodied" manner. However, we can use recursion to a similar effect. The code from this section can be found in ind.mlw.

Given a `p : int -> bool`, consider the following program:

```
let rec lemma ind (n : int) : unit =
requires { n >= 0 }
variant { n }
ensures { p n }
if n = 0 then () (* post: p 0 *)
else ind (n-1)   (* post: p (n-1) -> p n *)
```

This program does not return anything interesting (just the unit value). The construct `let rec lemma` will try to verify the recursive function function defining it and then extract the implication from precondition to postcondition as a lemma. Here, this implication would be

```
  forall n:int. n >= 0 -> p n
```

which is just the conclusion we try to establish by induction. Now let's examine the verification condition. In the then-branch of the conditional we have to prove

```
  forall n:int. n >= 0 /\ n = 0 -> p n
```

which amounts to proving `p 0`. In the else-branch of the conditional we have to establish the precondition, and then we can assume the postcondition of the recursive call to prove the postcondition. This become

```
forall n:int. n >= 0 /\ n <> 0 ->
  n-1 >= 0         (* precond. of rec. call *)
  p (n-1) -> p n   (* postcond. of rec. call implies postcond *)
```

With a little logic simplification this is equivalent to

```
forall n:int. n > 0 -> p (n-1) -> p n
```

which is the induction step.

In addition, Why3 also has to show that the argument n becomes smaller, that is,

```
forall n:int. n >= 0 /\ n <> 0 -> n-1 < n
```

which is clearly the case and does not depend on `p`.

Unfortunately, it turns out that we cannot simply parametrize the definition of `ind` with a predicate `p : int -> bool` because it seems Why3 can not perform the necessary higher-order reasoning over predicates. However, it can be packaged as a module, as explained in the next section.

Nonetheless, we can use this technique in specific cases, like proving by induction that

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

For the purpose of illustration, we define the sum on the left by axioms, and then the property we want explicitly as a predicate.

```
function sum(n:int) : int
axiom sum0 : sum 0 = 0
axiom sum1 : forall n. n >=0 ->
  sum (n+1) = sum n + (2*n+1)

predicate sum_odd (n:int) =
  n >= 0 -> sum n = n * n
```

At this point, proving that $\forall n.\, n \geq 0 \rightarrow \mathsf{sum\_odd}(n)$ would fail. So we set up the lemma

```
let rec lemma ind_sum_odd (n:int) : unit =
requires { n >= 0 }
variant { n }
ensures { sum_odd n }
if n = 0 then ()         (* sum_odd n *)
else ind_sum_odd (n-1)   (* sum_odd (n-1) -> sum_odd n *)
```

and Why3 can easily prove this. Just to confirm, we can set up a subsequent goal, which is also proved easily.

```
goal test : forall n. n >= 0 -> sum_odd n
```

This technique is quite flexible: if we need a different schema of induction, we can set up different lemmas. For example, if the base case is at $n = 1$ our precondition could

be $n \geq 1$, and if we go in steps of 2 we could write two base cases and subtract 2 from n before the recursive call.

As a short live-code example from lecture illustrating this pattern over lists, we proved that if two concatenations are equal and their prefixes have the same length, then the prefixes and suffixes must be equal. The lemma works by induction on the first list.

```
type word = list int

let rec lemma split_unique_ind (u1 u2 v1 v2 : word) =
  requires { length u1 = length u2 }
  requires { u1 ++ v1 = u2 ++ v2 }
  ensures { u1 = u2 /\ v1 = v2 }
  variant { u1 }
  match u1 with
  | Nil -> assert { u2 = Nil /\ v1 = v2 }
  | Cons a u1' ->
    match u2 with
    | Nil -> absurd
    | Cons b u2' -> assert { a = b } ; split_unique_ind u1' u2' v1 v2
    end
  end

goal split_unique : forall u1: word, u2: word, v1: word, v2: word .
  length u1 = length u2 ->
    u1 ++ v1 = u2 ++ v2 ->
      u1 = u2 /\ v1 = v2
```

Induction over inductive data types is an extremely useful tool in program verification.

5 Revisiting Dynamic Logic in Why3

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

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

We now continue our formalization of NDL in Why3. You can find the code in the file ndl-v2.mlw.

In order to represent the above definition, we define a new auxiliary predicate `run_bdd` and give the definition of $[\![\alpha^*]\!]$ in terms of the new predicate.

```
  predicate run (omega : state) (alpha : prog) (nu : state)
  predicate run_ind (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_ind omega alpha n nu
```

The formalization of `run_bdd` is via two simple axioms.

```
axiom run_ind_base : forall omega alpha nu .
  run_ind omega alpha 0 nu <-> omega = nu

axiom run_ind_n : forall omega alpha nu n . n > 0 ->
  run_ind omega alpha n nu <->
    exists mu . run omega alpha mu /\ run_ind mu alpha (n-1) nu
```

We were previously able to prove the loop-unrolling axiom of NDL.

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

Formalized as follows.

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

This no longer proves with the new inductive axioms for star. We will revisit this later, but for now we will focus on proving the representation of the NDL 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. What we want to show is the following inductive fact.

> *For all* $n \geq 0$ *and* ω,
> *if* $\omega \vDash Q$ *and* $\omega \vDash [\alpha^*](Q \rightarrow [\alpha]Q)$,
> *then for all* ν *where* $\omega[\![\alpha]\!]^n\nu$, $\nu \models Q$.

We prove this by induction on n. In Why3:

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

let rec lemma q_preserved_always (n: int) : unit =
  requires { 0 <= n }
  ensures { q_preserved n }
  variant { n }
  if n = 0 then ()
  else q_preserved_always (n - 1)
```

This lemma is strong enough to prove both directions of the induction axiom, so our work is done!

```
lemma box_ind : forall omega alpha q .
  models omega (Box (Star alpha) q) <->
    (models omega q /\
     models omega (Box (Star alpha) (Implies q (Box alpha q))))
```

Returning to the unrolling axiom from earlier, which failed after we changed our axioms to reflect the inductive form that we developed in lecture. Splitting the verification condition and inspecting which tasks Why3 is unable to prove, we find that it is only having difficulty with one case.

```
  ---------- Local Context ----------
  ...
  H1 : models omega q
  H  : models omega (Box alpha (Box (Star alpha) q))
  --------------- Goal --------------
  goal models_box_star : models omega (Box (Star alpha) q)
```

We find that we can use the same approach to establish the facts needed to prove unrolling as well, replacing the precondition with the corresponding antecedent from the direction of the axiom that fails to prove.

```
predicate box_q_preserved (n: int) = forall omega alpha q .
  models omega (Box (Star alpha) q) ->
    forall nu . run_ind omega alpha n nu ->
      models nu q

let rec lemma box_q_preserved_ind (n: int) : unit =
    requires { 0 <= n }
    ensures { box_q_preserved n }
    variant { n }
    if n = 0 then ()
    else box_q_preserved_ind (n - 1)
```

7 Induction with Invariants

Induction with invariants just requires one lemma regarding validity (as expressed with $\Box P$) and the properties we have already proved. We just summarize the lemma and two theorems here. These can be found at the end of the file ndl.mlw.

```
  (* lemma: Q /\ [](P -> [alpha]Q) -> [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: [](P -> Q) -> [alpha]P -> [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)

  (* induction with invariant J *)
  lemma induction_inv : forall omega j alpha q.
```

```
14    models omega (And j
15                  (And (Valid (Implies j (Box alpha j)))
16                       (Valid (Implies j q))))
17    -> models omega (Box (Star alpha) q)
```

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

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

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

8 Induction over Inductive Data Types

Instead of performing induction over `n : int` where $n \geq 0$, we can also define the natural numbers directly in unary form and perform induction over the structure of this representation. In this representation, we won't have all the usual operations on integers, but in come cases (like the proof the correctness of the axioms of NDL) they are not necessary and the prover may perform better.

As an example, here are the natural numbers and the expression of an induction principle over them in the form of a lemma as before.

```
1   type nat = Zero | Succ nat
2
3   let rec lemma ind_nat (n:nat) : unit =
4   variant { n }
5   ensures { P n }
6   match n with
7   | Zero -> ()              (* P Zero *)
8   | Succ n -> ind_nat n     (* forall n:nat. P n -> P (Succ n) *)
9 end
```

9 The SimpleInduction Module in Why3[1]

Among the Why3 standard libraries we find `int.SimpleInduction` for mathematical induction and also `int.Induction` for complete induction. This is less flexible than the explicit method, so we did not cover this in lecture.

```
1 module SimpleInduction
2   use Int
3   predicate p int
4   axiom base: p 0
5   axiom induction_step: forall n:int. 0 <= n -> p n -> p (n+1)
6   lemma SimpleInduction : forall n:int. 0 <= n -> p n
7 end
```

[1] not covered in lecture

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 SumSquare

  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
```