Lecture Notes on Induction

Frank Pfenning

Carnegie Mellon University
Lecture 9
Tuesday, February 15, 2022

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:

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

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.

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

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

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

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:

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

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.

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

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:

```
1 let rec lemma ind (n : int) : unit =
2 requires { n >= 0 }
3 variant { n }
4 ensures { p n }
5 if n = 0 then () (* post: p 0 *)
6 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

```
1    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

```
1    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

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

With a little logic simplification this is equivalent to

```
1    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,

```
1    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.

```
1    function sum(n:int) : int
2    axiom sum0 : sum 0 = 0
3    axiom sum1 : forall n. n >=0 ->
4      sum (n+1) = sum n + (2*n+1)
5
6    predicate sum_odd (n:int) =
7      n >= 0 -> sum n = n * n
```

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

```
1    let rec lemma ind_sum_odd (n:int) : unit =
2    requires { n >= 0 }
3    variant { n }
4    ensures { sum_odd n }
5    if n = 0 then ()          (* sum_odd n *)
6    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.

```
1    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.

At the end of this lecture (actually discussed at the beginning of the next lecture) we show how to apply this technique to perform induction over inductive data types.

5 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
```

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.

```
1   let rec function sum (a : int) (b : int) : int =
2   variant { b - a }
3   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.

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

We can now clone the standard library as sketched above.

```
1   clone int.SimpleInduction
2     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.

---

[1]not covered in lecture

```
1  theory SumSquare
2
3    use int.Int
4    use int.EuclideanDivision
5
6    let rec function sum (a : int) (b : int) : int =
7    variant { b - a }
8    if a > b then 0 else sum a (b-1) + b
9
10   predicate sum_square (n : int) =
11     n >= 0 -> sum 0 n = div (n*(n+1)) 2
12
13   clone int.SimpleInduction
14     with predicate p = sum_square, lemma base, lemma induction_step
15
16   goal G : forall n:int. n >= 0 -> sum_square n
17
18 end
```

6 Revisiting Dynamic Logic in Why3

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

$$\omega[\![\alpha^*]\!]\nu \qquad \text{iff there exists an } n \geq 0 \text{ such that } \omega[\![\alpha]\!]^n\nu$$

$$\omega[\![\alpha]\!]^0\nu \qquad \text{iff } \omega = \nu$$
$$\omega[\![\alpha]\!]^{n+1}\nu \quad \text{iff there exists } \mu \text{ such that } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\alpha]\!]^n\nu$$

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.

```
1    predicate run (omega : state) (alpha : prog) (nu : state)
2    predicate run_bdd (omega:state) (alpha:prog) (n:int) (nu:state)
3
4    axiom run_star : forall omega alpha nu.
5      run omega (Star alpha) nu
6      <-> exists n. n >= 0 /\ run_bdd omega alpha n nu
```

The formalization of run_bdd is via two simple axioms.

```
1    axiom run_bdd_0 : forall omega alpha nu.
2      run_bdd omega alpha 0 nu <-> omega = nu
3    axiom run_bdd_1 : forall omega alpha n nu. n > 0 ->
4      run_bdd omega alpha n nu
5      <-> exists mu. run omega alpha mu /\ run_bdd mu alpha (n-1) nu
```

We'd now like to show that this implies the recursive specification, namely that

$$\omega[\![\alpha^*]\!]\nu \text{ iff } \omega = \nu \lor \exists\mu.\, \omega[\![\alpha]\!]\mu \land \mu[\![\alpha^*]\!]\nu$$

Unfortunately, Why3 could not establish this directly, but we needed two simple lemmas. We conjecture that the existential quantification over $n \geq 0$ in the definition of $[\![\alpha^*]\!]$ is to blame. In any case, with those two lemmas we can prove the above characterization.

```
1  lemma run_bdd_left0 : forall omega alpha.
2    run omega (Star alpha) omega
3
4  lemma run_bdd_left1 : forall omega alpha mu n nu. n >= 0 ->
5    run omega alpha mu /\ run_bdd mu alpha n nu
6    -> run_bdd omega alpha (n+1) nu
7
8  lemma run_star_rec : forall omega alpha nu.
9    (omega = nu
10    \/ exists mu. run omega alpha mu /\ run mu (Star alpha) nu)
11    <-> run omega (Star alpha) nu
```

Similarly, the loop unrolling axiom of dynamic logic, namely

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

is easy to prove in both directions.

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

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

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

7 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:

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

```
4
5    let rec lemma ind_run_bdd_left (n:int) : unit =
6    requires { n >= 0 }
7    variant { n }
8    ensures { run_bdd_left  n }
9    if n = 0 then () else ind_run_bdd_left (n-1)
```

We can similarly prove the opposite direction of this implication, but it does not appear to be necessary: We can already prove one direction of the induction axiom without any further induction.

```
1    lemma induction_unwind : forall omega alpha q.
2      models omega (Box (Star alpha) q)
3      -> models omega q
4      /\ models omega (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.

```
1    predicate run_bdd_inv (n:int) = forall omega alpha q.
2      models omega q
3      /\ models omega (Box (Star alpha) (Implies q (Box alpha q)))
4      -> forall nu. run_bdd omega alpha n nu -> models nu q
5
6    let rec lemma ind_run_bdd_inv (n:int) : unit =
7    requires { n >= 0 }
8    variant { n }
9    ensures { run_bdd_inv n }
10   if n = 0 then () else ind_run_bdd_inv (n-1)
11
12   lemma induction_wind : forall omega alpha q.
13   models omega q
14   /\ models omega (Box (Star alpha) (Implies q (Box alpha q)))
15   -> models omega (Box (Star alpha) q)
```

8 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-v2.mlw.

```
1    (* lemma: Q /\ [](P -> [alpha]Q) -> [alpha*]Q *)
2    lemma induction_valid : forall omega q alpha.
3     models omega (And q (Valid (Implies q (Box alpha q))))
4     -> models omega (Box (Star alpha) q)
5
6    (* lemma: [](P -> Q) -> [alpha]P -> [alpha]Q *)
7    lemma dist_valid_box : forall omega alpha p q.
8     models omega (Valid (Implies p q))
9     -> models omega (Box alpha p)
10    -> models omega (Box alpha q)
11
12   (* induction with invariant J *)
```

```
13    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 $\langle\alpha^*\rangle Q$.

9 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
```

In an alternative development of NDL, using natural numbers to express iteration, we can avoid some puzzling lemmas from the development presented earlier. You can find the alternative development in the file ndl-nat.mlw.