15-414: Bug Catching: Automated Program Verification

Lecture Notes on Arrays

Matt Fredrikson

Carnegie Mellon University

Lecture 9
September 23, 2025

1 Introduction

Our efforts to formalize how we can prove the correctness of programs have paid off! We have presented a formal language of programs α and their semantics as a relation $\omega[\![\alpha]\!]\nu$ between a prestates ω and a poststates ν. States here map variables to their integer values. We then also formalized a language of formulas P in dynamic logic and when they are true in a state, $\omega \models P$. A characteristic of dynamic logic are three modal operators $[\alpha]P$ (after all possible runs of α, the formula P is true), $\langle\alpha\rangle P$ (there is a run of α after which P is true), and $\Box P$ (P is valid, that is, true in all possible states). Finally, we have provided some axioms for reasoning about programs in dynamic logic in a rigorous way. These axioms allow us to break down the structure of programs occurring in formulas until we are left with a formula in integer arithmetic, no longer containing $[\alpha]P$ and $\langle\alpha\rangle P$. This required *loop invariants* in order eliminate $[\alpha^*]P$. As we will see in the next lecture, we also need *loop variants* to eliminate $\langle\alpha^*\rangle P$.

After eliminating programs, we end up with a formula in the *language of integer arithmetic*. Most programs we write are not confined to integers but use other data types such as lists, trees, arrays, streams, etc. How do we integrate such types into the formal reasoning framework? On one hand, integers are universal in that there is a technique called *Gödel numbering* by which we code elements of these other types as integers. On the other hand, this is only of theoretical interest in that it is not practical to reason, say, about trees using their Gödel numbers. Mathematicians have long established the *axiomatic method* to reason about complex structures. We keep matters abstract by defining a language that denotes elements of the structure and then axioms we can use to reasoning about them. For example, the theory of groups arises from postulating a binary operation '$a \cdot b$', an unary inversion 'a^{-1}', and a neutral element e and some axioms such as $a \cdot a^{-1} = e$. An important consequence of this approach is that whenever we recognize that a concrete structure satisfies the axioms then we can inherit all the theorems we have proven for this instance.

We will follow the axiomatic approach, namely specifying data structures with new operations and axioms describing their properties. Depending on how common particular data types are, back-end provers may either treat them generically (reasoning directly with the axioms) or use specialized decision procedures for particular theories. One of the big research questions in this area is then how to combine theories in a sound and perhaps even complete way. In imperative programming, a particularly important theory is the *theory of arrays* because they constitute such a widely used abstraction in imperative programming.

We have already implicitly applied this approach when we implemented and verified a regular expression matcher in Lecture 6. In this lecture, we will study dynamic logic itself which (suprisingly!) provides a nice illustration of the theory of arrays.

An alternative approach is that of type theory, which provides general mechanisms to define large classes of types such as inductive types (like lists, trees, etc.), coinductive types (like streams), quotient types (like rational numbers), etc. In this approach we obtain programming constructs (like pattern matching against trees) and matching reasoning principles (like induction over trees) from the same definitions. This approach is studied in courses such as 15-312 *Foundations of Programming Languages* and 15-317 *Constructive Logic*.

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

- Use read-over-write axioms to reason about (abstract) arrays
- Use extensionality to reason about array equality
- Apply the axiomatic method to new theories
- Reason, at the meta-level, about dynamic logic in Why3

2 The Theory of Arrays

How do we reason about arrays in an imperative language? Because arrays are a mutable data structure this is actually quite tricky, as you likely have experienced in the programming assignments. When we write to an array at a given index, for example, we have to keep track not only that this particular element changes but that all others remain the same. Consequently, loop invariants are complicated because they have to address not only how arrays are changed but also how arrays do not change. Moreover, "change" is really not something that is easily modeled in logic—witness, all the effort we had to go through in our development of dynamic logic. At first one might think we can just provide some new constructs in dynamic logic for programs that perform, for example, array assignment. But that's not enough: we also need to be able to eliminate such programs so we can reason about the resulting pure formula in the theory of arrays.

A key step, then is to abstract away from the imperative nature of arrays to an abstract axiomatic form. For that purpose we have a type ι for array indices and a separate type τ for the array elements, as well as a type $\mathsf{array}\,\iota\,\tau$ for arrays. We then have just two operations, read and write.

$\mathsf{read}\,a\,i : \tau$	read from array a at index i
$\mathsf{write}\,a\,i\,v : \mathsf{array}\,\iota\,\tau$	write to array a at index i

The second expression does not mutate a (a concept that doesn't directly exist) but denotes a new array distinct from a.

There are just two axioms defining these operations

$$\begin{array}{l} i = k \rightarrow \mathsf{read}\,(\mathsf{write}\,a\,i\,v)\,k = v \\ i \neq k \rightarrow \mathsf{read}\,(\mathsf{write}\,a\,i\,v)\,k = \mathsf{read}\,a\,k \end{array}$$

These axioms are referred to as *read-over-write* since the read operation is applied to the result of a write operation. We do not directly reason about whether "array access is in bounds" which we delegate to the translation from the concrete arrays in the programming language to abstract arrays.

As we will see when we implement this theory, these axioms will allow us to prove a lot of useful properties, but by themselves are insufficient. The problem is exemplified the the following equation:

$$\mathsf{write}\,a\,i\,(\mathsf{read}\,a\,i) \overset{?}{=} a$$

If we write back to an array at index i the value already contained in the array at that index, we should get back an equal array. We might try to think of many ways to add axioms to capture this property, but there is a generic and powerful way to do this using the *axiom of extensionality* for arrays.

$$(\forall i.\, \mathsf{read}\,a\,i = \mathsf{read}\,a'\,i) \leftrightarrow a = a'$$

Why does this capture our example above? We have to prove

$$\forall k.\, \mathsf{read}\,(\mathsf{write}\,a\,i\,(\mathsf{read}\,a\,i))\,k \overset{?}{=} \mathsf{read}\,a\,k$$

We distinguish two cases: $k = i$ and $k \neq i$. If $i = k$, we simplify the left-hand side using the *first* read-over-write axiom to $\mathsf{read}\,a\,i$ which is equal to $\mathsf{read}\,a\,k$ since $i = k$. If $k \neq i$, we simplify the left-hand side using the *second* read-over-write axiom to $\mathsf{read}\,a\,k$, which is the same as the right-hand side.

3 Translating Mutable Arrays to Logical Arrays

We sketch how mutable arrays in programs are translated to logical arrays in the theory of arrays. Essentially, an expression $a[e]$ requires a precondition that $0 \leq e < \mathsf{length}\,a$

and becomes $\mathsf{read}\, a\, e$. An assignment $a[e] \leftarrow e'$ also generates a precondition that $0 \leq e < a.\mathsf{length}$ and then becomes $a' \leftarrow \mathsf{write}\, a\, e\, e'$, where a' is substituted for a in the remaining program and postcondition. For this purpose we need to add a new constant $a.\mathsf{length}$ add some axioms, such as $(\mathsf{write}\, a\, i\, v).\mathsf{length} = a.\mathsf{length}$.

Because of the preconditions, we separate out array accesses into a new form of program

$$x \leftarrow a[e]$$

For example,

$$a[i] \leftarrow a[i] + 1$$

would become

$$\begin{array}{l} x \leftarrow a[i] \\ a[i] \leftarrow x + 1 \end{array}$$

for a fresh x.

We can describe in the form of axioms in dynamic logic how we reason about the new constructs.

$$[x \leftarrow a[e]]P(x) \leftrightarrow 0 \leq e \wedge e < a.\mathsf{length} \wedge \forall x.x' = \mathsf{read}\, a\, e \rightarrow P(x') \qquad (x' \text{ not in } a[e] \text{ or } P(x))$$

$$[a[e] \leftarrow e']P(a) \leftrightarrow 0 \leq e \wedge e < a.\mathsf{length} \wedge \forall a'.a' = \mathsf{write}\, a\, e\, e' \rightarrow P(a') \qquad (a' \text{ not in } a[e], e' \text{ or } P(a))$$

4 States as Arrays

After a moment's reflection you should be able to see that *states* that we have been using to define dynamic logic use exactly the operations for abstract arrays! States are indexed by variables that are mapped to integers $c \in \mathbb{Z}$.

$\omega(x)$	$\mathsf{read}\, \omega\, x$
$\omega[x \mapsto c]$	$\mathsf{write}\, \omega\, x\, c$

We didn't need anything else.

This now suggests a potentially ambitious project: could we formalize the (abstract) syntax and semantics of dynamic logic?

The rest of this lecture will be devoted to give a positive answer to this question.

A key decision is to focus on *reasoning about programs* rather than *executing* them. We therefore don't even provide a concrete definition of variables and states—they remain *abstract*. The only thing we know about them are the read-over-write and extensionality axioms. We do, however, commit to values being integers so we can reason about some concrete programs manipulating integers. Note, specifically, that types `state` and `var` are declared but not defined.

```
module NDL

  use int.Int
```

```

type state
type var

(* "array" operations and axioms *)
function read (omega : state) (x : var) : int
function write (omega : state) (x : var) (v : int) : state

axiom read_eq : forall x y omega v.
  x = y -> read (write omega x v) y = v
axiom read_ne : forall x y omega v.
  x <> y -> read (write omega x v) y = read omega y

(* extensionality *)
axiom ext : forall omega nu.
  (forall x. read omega x = read nu x) <-> omega = nu

end
```

We test our encoding by letting Why3 verify a simple property of arrays mentioned before: when we write back the element we just read the array will be unchanged.

```
goal test1 : forall a i.
  write a i (read a i) = a
```

Fortunately, this can be proved without any problems. We say here `goal` which means it must be proved in a verification effort but the fact will not be used subsequently. Doing so is not logically problematic, but the more axioms and lemmas there are, the more difficult the theorem proving effort so it is generally best to introduce lemmas only when they are needed.

5 Expressions and Evaluation

Next step will be to define expressions. We use the facility to define recursive types in Why3, which we have seen before for regular expressions and some homework problems.

Evaluation will then be a *function* from a state and an expression to an integer. We declare it with `function` instead of `let function` since we do not intend to use it computationally.

```
type exp = Const int | Var var | Plus exp exp | Minus exp exp (*...*)

function eval (omega : state) (e : exp) : int =
  match e with
  | Const v -> v
  | Var x -> read omega x
  | Plus e e' -> (eval omega e) + (eval omega e')
  | Minus e e' -> (eval omega e) - (eval omega e')
  end
```

For a more realistic language we would have further kinds of expressions, but for this small experiment it is enough to include variables, integer constants, addition, and subtraction. Again, we test our definition.

```
goal test2 : forall omega x .
  read omega x = 0 -> eval omega (Plus (Var x) (Const 1)) = 1
```

6 Programs and Propositions

In dynamic logic, propositions and programs mutually depend on each other. We therefore define them as mutually recursive types using the `with` keyword of WhyML. Programs will have type `prog` and propostions type `ndl` (for nondeterministic dynamic logic). The mutual dependency lies in the `Guard`, `Box` and `Dia` constructs.

Note that we do not define quantifiers in our `ndl` type. Doing so would require defining the rules of variable substitution, which is a surprisingly intricate (and labor-intensive) exercise. Because of this, we will be unable to prove the axiom for variable assignment in our formalization. However, this is not essential for our purposes in this and the following lecture.

```
type prog =
  Assign var exp
| Sequence prog prog
| Test ndl
| Union prog prog
| Star prog
with ndl =
  Equal exp exp
| Not ndl
| And ndl ndl
| Or ndl ndl
| Implies ndl ndl
| Box prog ndl
| Dia prog ndl
```

Next, we come to the meaning of programs. Again for simplicity we use the nondeterministic version of dynamic logic programs; conditionals and while loops can be coded in standard way we discussed earlier, as can be skip and abort.

$$\begin{array}{lcl}
\mathsf{skip} & \triangleq & ?\mathsf{true} \\
\mathsf{abort} & \triangleq & ?\mathsf{false} \\
\mathsf{if}\, P\, \alpha\, \beta & \triangleq & (?P\ ;\ \alpha) \cup (?\neg P\ ;\ \beta) \\
\mathsf{while}\, P\, \alpha & \triangleq & (?P\ ;\ \alpha)^*\ ;\ ?\neg P
\end{array}$$

An explicit definition of $\omega[\![\alpha]\!]\nu$ is rejected by Why3 since it cannot prove termination (and should not be able to prove termination) due to repetition. We therefore turn each clause in the usual semantic definition into an axiom. For example,

$$\omega[\![x \leftarrow e]\!]\nu \quad \text{iff } \nu = \omega[x \mapsto \omega[\![e]\!]]$$

is turned into

```
predicate run (omega : state) (alpha : prog) (nu : state)

axiom run_assign : forall omega x e nu.
  run omega (Assign x e) nu <-> write omega x (eval omega e) = nu
```

Here are the remaining axioms.

```
axiom run_assign : forall omega nu x e .
  run omega (Assign x e) nu <-> nu = write omega x (eval omega e)
axiom run_sequence : forall omega alpha beta nu .
  run omega (Sequence alpha beta) nu <-> exists mu . (run omega alpha
      mu) /\ (run mu beta nu)
axiom run_test : forall omega p nu .
  run omega (Test p) nu <-> omega = nu /\ models omega p
axiom run_union : forall omega alpha beta nu .
  run omega (Union alpha beta) nu <-> run omega alpha nu \/ run omega
      beta nu
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
```

By necessity, the last one requires the `models` predicate where `models omega p` iff $\omega \models P$.

7 Formulas

Next, we come to the formulas P of dynamic logic. We specify the semantics $\omega \models P$ with a corresponding predicate in WhyML. The semantic definitions then become axioms. For the ordinary logical connectives, we just translate them to the corresponding connectives in the metalogic of WhyML. For the modal operators $[\alpha]Q$ and $\langle\alpha\rangle Q$ we use their definitions in terms of all or some possible poststates.

```
axiom models_equals : forall omega e e' .
  models omega (Equal e e') <-> (eval omega e) = (eval omega e')
axiom models_not : forall omega p .
  models omega (Not p) <-> not (models omega p)
axiom models_and : forall omega p q .
  models omega (And p q) <-> (models omega p) /\ (models omega q)
axiom models_or : forall omega p q .
    models omega (Or p q) <-> (models omega p) \/ (models omega q)
axiom models_implication : forall omega p q .
  models omega (Implies p q) <-> ((models omega p) -> (models omega q)
      )
axiom models_box : forall omega alpha q .
  models omega (Box alpha q) <-> forall nu . run omega alpha nu ->
      models nu q
axiom models_dia : forall omega alpha q .
  models omega (Dia alpha q) <-> exists nu . run omega alpha nu /\
      models nu q
```

We can now test our semantics.

```
goal test4 : forall omega x .
```

```
  models omega (Dia (Union (Assign x (Const 1))
                           (Assign x (Const 2)))
                    (Equal (Var x) (Const 1)))
```

Unfortunately, Why3 is unable to prove it! We can address this by writing a "let" lemma: a ghost function whose postcondition expresses the fact that we want Why3 to prove. This allows us to break the proof goal down, and importantly in this case, show Why3 that the diamond formula is true by constructing a final state of the program that satisfies $x = 1$.

```
let lemma test4_explicit (omega : state) (x : var) : unit =
  ensures { models omega (Dia (Union (Assign x (Const 1)) (Assign x (
      Const 2))) (Equal (Var x) (Const 1))) }
  let alpha1 = (Assign x (Const 1)) in
  let alpha2 = (Assign x (Const 2)) in
  let q = (Equal (Var x) (Const 1)) in
  let nu = write omega x 1 in
  assert { models nu q };
```

Let-lemmas are a useful tool for proving statements that Why3 is unable to handle automatically. We will see more examples of this in the next lecture, when we discuss induction.

8 Proving the Axioms

We developed some axioms in NDL so we could reduce properties of programs to arithmetic formulas only, in a syntactic manner. While they are axioms in dynamic logic, we proved that they are valid with respect to the semantic interpretation of the formulas. Actually, we proved only one or two but convinced ourselves that we could prove the others.

As a simple example, consider the axiom to break down program composition.

$$[\alpha\,;\,\beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

We express this as a `lemma` (which means Why3 will try to prove this from the preceding axioms and definitions).

```
  lemma models_seq : forall omega alpha beta q.
    models omega (Box (Seq alpha beta) q)
    <-> models omega (Box alpha (Box beta q))
```

Here we converted the '$\leftrightarrow$' into a `<->` at the metalevel instead of an explicit dynamic logic connective.

Turns out that Why3 can easily prove the soundness of this axiom, and the other ones except assignment (which we skipped due to the complication of variable renaming) and induction (since, as the name suggests, would require induction). We will return to induction in the next lecture.

```
lemma box_test : forall omega p q .
  models omega (Box (Test p) q) <->
```

```
    models omega (Implies p q)

lemma box_union : forall omega alpha beta q .
  models omega (Box (Union alpha beta) q) <->
    ((models omega (Box alpha q)) /\ (models omega (Box beta q)))

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

The complete live code from this lecture can be found in the file ndl-v1.mlw.

9 Discussion

What we have done today is actually a bit mind-bending. We spent several lectures to develop dynamic logic in order to justify the correctness of what Why3 was doing. Or at least some abstraction of it that was tractable. In this lecture we used Why3 in order to formalize and, to some extent, verify the axioms to reason about programs in dynamic logic.

Besides being cool, it should help increase our confidence in dynamic logic, even though there are possible loopholes: if Why3 is incorrect because its foundations are wrong, then Why3 might certify dynamic logic even though it is also wrong. But we also saw in lecture that if we made a mistake in any of the proposed axioms, they were not proved. So it is clearly better than simply relying on our general ability to reason mathematically. Ultimately, though, we can never be completely certain in this kind of metacircular formalization. That would be too much to ask.

There are techniques to reduce what we call the *trusted computing base*, something we can justify mathematically but not within the programming language. For example, if we asked Why3's decision procedures to hand us a formal proof (rather than just saying *yes*, *no*, or *maybe*) we could then independently check the validity of the proof with a small checker. Such so-called *proof-producing decision procedures* are quite interesting and possibly the subject of a lecture later in the semester, once we have covered the notion of *formal proof*.