15-414: Bug Catching: Automated Program Verification

Lecture Notes on Data Structures and Invariants

Matt Fredrikson*

Carnegie Mellon University

Lecture 11

1 Introduction

What this course studies so far is a pretty comprehensive account of the reasoning principles for algorithms implemented in imperative programs.[1] But the point of today's lecture is that there is more to understanding programs than just understanding their algorithms. Algorithms and data structures courses have two major components: first the algorithms and then the data structures. Indeed, a lot of the more advanced insights come from algorithms that store their data in ways that are particularly well-suited for the task at hand. Just think of how much faster it is to find an entry in an array that is at least sorted compared to naive linear search in an unsorted array. Or remind yourself how important it was that data is stored and shuffled around again in the right way when using various tree data structures such as AVL trees to enable fast searching and insertion.

This lecture will, thus, explore how data structures impact our reasoning.

2 Recap: Deterministic and Nondeterministic Repetitions

Recall the proof rule that proves properties of while loops using loop invariants:

$$\text{(while)}\ \frac{\Gamma \vdash J, \Delta \quad J, Q \vdash [\alpha]J \quad J, \neg Q \vdash P}{\Gamma \vdash [\mathtt{while}(Q)\, \alpha]P, \Delta}$$

*Based on lecture notes written by Andre Platzer

[1] The reasoning principles investigated in this course extend to object-oriented programming languages, both core languages [BP06] and real languages such as Java in the verification tool KeY [ABB$^+$16].

Recall the nondeterministic repetition α^* which expresses that the program α repeats any arbitrary unspecified nondeterministic number of times:

6. $[\![\alpha^*]\!] = \big\{(\omega,\nu)\ :\ $ there are an n and states $\mu_0 = \omega, \mu_1, \mu_2, \dots, \mu_n = \nu$ such that $(\mu_i, \mu_{i+1}) \in [\![\alpha]\!]$ for all $0 \leq i < n\}$
   That is, state μ_{i+1} is reachable from state μ_i by running α for all i.

Recall its core induction principle for $[\alpha^*]P$.

Lemma 1. *The induction axiom I is sound:*

$$(I)\quad [\alpha^*]P \leftrightarrow P \land [\alpha^*](P \rightarrow [\alpha]P)$$

The loop invariant rule for nondeterministic repetitions derives from this axiom:

$$\text{(loop)}\ \frac{\Gamma \vdash J, \Delta \quad J \vdash [\alpha]J \quad J \vdash P}{\Gamma \vdash [\alpha^*]P, \Delta}$$

3 A Silly Data Structure

For illustration purposes, let's consider the world's most trivial data structure: A data structure remembering a bit and providing operations to flip, set, or check the value of the bit.[2]

```
type bit;
bit init();
bit flip(bit b);
bit set(bit b);
bool check(bit b);
```

The bit data structure with this interface can be implemented with an integer for data storage in a straightforward way:

```
type bit          = int;
bit init()        { return 0; }
bit flip(bit b)   { return 1-b; }
bit set(bit b)    { return 1; }
bool check(bit b) { return (b==1); }
```

What do we need to check to make sure this implementation works correctly?

This data structure implementation is so simple, it doesn't look like there'd be anything worth checking for it. But, nevertheless, if we want to get it right, we might as well worry about what we need to do to make sure it's been implemented correctly.

[2] If you want the bit data structure to not be quite so entirely dumb, imagine it being part of implementing a semaphore correctly.

4 What Data Structures Want

What could go wrong in the above data structure implementation? Flipping a bit via subtraction `1-b` will only flip the bit correctly to 1 or 0 if it was previously 0 or 1. Otherwise `1-b` will lead to whatever arbitrary values. Consequently, the integer implementation of the bit data structure assumes that this formula is maintained as a data structure invariant all the time:

$$b = 0 \lor b = 1$$

Only if the integer representing the bit always is either 0 or 1, will the operation `1-b` flip it correctly. What does it mean to have such a data structure invariant? When exactly does it need to be true? How can we prove that it is?

5 Data Structures and Their Clients

We could set out proving the data structure invariant holds true everywhere, but that will not actually be the case without additional assumptions. We could also prove and assume it in some places that make sense to us, but will still be left with the odd feeling that we are not entirely sure whether we've really assumed and proved it in the right places. After all if we happily assume the data structure invariant in a position where we should have proved it, then we haven't done much good.

Instead, we will systematically derive what it means to be a data structure invariant from our previous understanding of invariant reasoning about code. A data structure such as the bit data structure has a present state and is being operated on by calling its flip, set, or check operations. Of course, before being able to call any operation on a bit data structure, we will first have to initialize an instance of the data structure by calling `init()` to obtain a bit in the first place. But after we have such a bit, let's call it d, then any number of its operations can be called in any order.

If we had a fixed client program for the data structure, then we could verify the correctness of this particular client program that is calling functions of the data structures using the inlining or other procedure call proof rules.

But, of course, we don't have such a client program and would even like to understand and verify the correctness of the bit data structure independently of how it is being used by any particular client program.

Yet, there is still something fundamental and quite generically useful to be understood about the intended way of using data structures even without any specific client program in mind. Clients will first have to initialize the data structure and can then call any number of operations on it in any arbitrary order. We cannot predict what particular operations the client will call in which order. But we can express that if we continue the power of nondeterminism.

6 Nondeterministic Assignments

Let's introduce a new program statement for the purpose of nondeterministically choosing an arbitrary integer value and assigning it to the variable x:

$$x := *$$

It's semantics is

$$[\![x := *]\!] = \{(\omega, \nu) \ : \ \nu = \omega \text{ except on variable } x\}$$

In fact, nondeterministic assignments can already be defined from nondeterministic repetitions as:

$$\{x := x + 1\}^*$$

Oh, no wait a moment, this program increments x any number of times, but that merely increases x, so it can never come out smaller than it already was initially. Let's try that again:

$$\{x := x + 1\}^*; \ \{x := x - 1\}^*$$

This program will give x an arbitrary integer value just like $x := *$ is supposed to.

Proving that a property P is true after all ways of running $x := *$ amounts to showing that P is true for all values of x, because any of them might be assigned to x after running $x := *$:

$$([:*]) \quad [x := *]P \leftrightarrow \forall x\, P$$

7 An Operational Model of the Use of a Data Structure

With these preparations, it is now easy to directly express how general clients are supposed to use a data structure without any particular assumptions on how exactly they do so. Here is the operational model for using the (impoverished) bit data structure:

```
$d := \text{init}();$
$p := 0;$
while$(p \geq 0)\,\{$
   $p := *;$
   if$(p = 0)\ d := \text{flip}(d)$
   else if$(p = 1)\ d := \text{set}(d)$
   else if$(p = 2)\ x := \text{check}(d)$
$\}$
```

This program expresses that the bit data structure first has to be initialized with `init()` to even obtain an instance d of a bit. As long as the operation selection variable p is nonnegative, an arbitrary integer is placed into the operation selection variable p nondeterministically by $p := *$, whose value then determines which particular operation is performed on the data structure.

0. If $p = 0$ then the data structure bit is flipped via calling `d:=flip(d)`

1. If $p = 1$ then the data structure bit is set via calling `d:=set(d)`

2. If $p = 2$ then the data structure value is queried and stored in a variable x via calling `x:=check(d)`

The data structure stops being used whenever p becomes negative.

It is not hard to convince yourself informally why any particular use of the bit data structure would correspond to a particular sequence of choices for the nondeterministic assignment $p := *$ of the operation selection variable. In a real client use case, the next choice of the operation may well depend on the result of the queried variable x from a check, but that's still just a matter of resolving the nondeterministic assignment appropriately.

8 The Meaning of Data Structure Invariants

Having rendered the intended operational model of the use of a data structure as a (nondeterministic) program, we can now make sense of what a data structure invariant is supposed to be. A *data structure invariant* is simply a formula J that can be proved to be a loop invariant for the corresponding operational model of its use as a nondeterministic program.

In the case of the bit data structure, its data structure invariant is

$$J \overset{\text{def}}{\equiv} d = 0 \lor d = 1 \tag{1}$$

Since (as usual) the operation selection variable p does not occur in the data structure invariant J, it quickly vanishes from the proof by appropriate weakening. Suitable use of the axioms and proof rules [;],$[:=]_=$,while,[if] as well as axiom [:$*$] followed by rule $\forall$R leads to the following:

$$\frac{\vdash [d := \text{init}()]J \quad J \vdash [d := \text{flip}(d)]J \quad J \vdash [d := \text{set}(d)]J \quad J \vdash [x := \text{check}(d)]J}{\vdash [d := \text{init}(); p := 0; \texttt{while}(p \geq 0)\,\{p := *; \texttt{if}(p = 0)\, d := \text{flip}(d) \texttt{else if}(p = 1)\, d := \text{set}(d) \texttt{else} \ \ldots\}]J}$$

The proof splits off the initialization program with [;],$[:=]_=$ leading to the first remaining premise. The appropriate loop invariant for while is J from (1) with a trivial use case. Then, after suitable [;] splitting, the proof considers any arbitrary value of operation selection variable p with [:$*$],$\forall$R. Splitting cases with [;],[if] in the induction step leads to the second, third, and fourth remaining premise after suitable weakening to discard the useless assumptions about the corresponding values of p.

Finally observe that the last premise

$$J \vdash [x := \text{check}(d)]J$$

proves trivially, because the data structure d, which is the only free variable of J, does not change in $x := \text{check}(d)$. This is a general property of pure functions that merely retrieve information from the data structure without changing it. For those, it is obvious that they cannot accidentally damage the data structure invariant. In fact, the fourth premise is easily provable using the vacuous axiom V

$$(\text{V})\;\; p \to [\alpha]p \quad (FV(p)\cap BV(\alpha)=\emptyset)$$

This leads to just three premises that still need to be proved to show that J is, indeed, a data structure invariant for the bit data structure:

$$\frac{\vdash [d:=\text{init}()]J \quad J\vdash [d:=\text{flip}(d)]J \quad J\vdash [d:=\text{set}(d)]J}{\vdash [d:=\text{init}();p:=0;\texttt{while}(p\geq 0)\,\{p:=*;\texttt{if}(p=0)\,d:=\text{flip}(d)\texttt{else if}(p=1)\,d:=\text{set}(d)\texttt{else }\ldots\}]J}$$

After inlining / macro expansion, all those proofs are obvious thanks to the data structure invariant.

$$\text{[inl]}\;\dfrac{\text{[:=]},\mathbb{Z}\;\dfrac{*}{\vdash [d:=0](d=0\lor d=1)}}{\vdash [d:=\text{init}()]J}$$

$$\text{[inl]}\;\dfrac{\text{[:=]},\mathbb{Z}\;\dfrac{*}{d=0\lor d=1\vdash [d:=1-d](d=0\lor d=1)}}{J\vdash [d:=\text{flip(d)}]J}$$

$$\text{WL,[inl]}\;\dfrac{\text{[:=]},\mathbb{Z}\;\dfrac{*}{\vdash [d:=1](d=0\lor d=1)}}{J\vdash [d:=\text{set(d)}]J}$$

Observe, of course, that weakening to discard the data structure invariant was only possible in the `set()` function, not in the `flip()` function, whose correctness crucially depends on d only ever being 0 or 1.

Note how loop invariant reasoning techniques have now made it perfectly precise in what way data structure invariants are invariants, and what it means for them to be preserved by the data structure.

9 Contracts for Data Structure Operations

Now that we have succeeded with establishing a correctness proof for the data structure invariant (1) for the bit data structure, it is about time to wonder what a user of the data structure can gain from it. Nothing at all! The reason is that the data structure invariant is a purely internal aspect that's only relevant to the correct implementation of the data structure not to its use from the outside. If the data structure ever breaks, then that's the fault of the data structure's implementation, not the client's. If there are any particular requirements on when the client is allowed to call what operation on the data structure, then those should have been expressed as contracts on its interface right away.

That does not happen for the simple bit data structure because any operation can be called on it. However, if we want to use a bit to implement a semaphore, we might also need to add a function lower that we are only allowed to call if the bit is currently set (and will then try to clear the bit):

```
//@requires check(d);
bit lower(bit d);
```

Such a contract can be incorporated into the operational model for the data structure by just making sure that the `lower()` function can only be called when its precondition is satisfied:

```
$d := \text{init}();$
$p := 0;$
while$(p \geq 0)$ {
   $p := *;$
   if$(p = 0)\ d := \text{flip}(d)$
   else if$(p = 1)\ d := \text{set}(d)$
   else if$(p = 2)\ x := \text{check}(d)$
   else if$\boldsymbol{(p = 3 \wedge \text{check}(d))\ d := \text{lower}(d)}$
}
```

Again, nondeterminism saves the day by just skipping the function call when its precondition is not satisfied and the repeating the loop with another selection of $p := *$. Because of the additional conjunct in the if-condition, the required precondition check(d) will be available in the corresponding proof of the data structure invariant:

$$J, \text{check}(d) \vdash [d := \text{lower}(d)]J$$

10 Postconditions for Data Structure Operations

Postcondition contracts can be established separately as usual for the procedures operating on a data structure. Postcondition checking can also be incorporated into the same framework of understanding data structure invariants by reasoning logically about their operational model. It does lead to some additional encoding that enters a failed state (which invalidates the data structure invariant) whenever a postcondition contract is not satisfied. One can also add an assert statement that terminates execution failing all postconditions unless its condition is met.

Suppose we augment the contract for the lower function with a postcondition:

```
//@requires check(d);
//@ensures !check(\result);
bit lower(bit d);
```

This leads to a slightly modified operational model for the use of the data structure:

```
$d := \text{init}();$
$p := 0;$
while$(p \geq 0)\ \{$
  $p := *;$
  if$(p = 0)\ d := \text{flip}(d)$
  else if$(p = 1)\ d := \text{set}(d)$
  else if$(p = 2)\ x := \text{check}(d)$
  else if$(\boldsymbol{p = 3} \land \text{check}(\boldsymbol{d}))\ \boldsymbol{\{d :=} \text{ lower}(\boldsymbol{d});$ assert$(\neg\text{check}(\boldsymbol{d})\boldsymbol{\}}$
$\}$
```

11 When Data Structure Invariants are True

All variants of the operational model of data structure uses that this lecture considered as well as the resulting sequent calculus premises made it crystal clear when exactly data structure invariants are expected to be true. Right after initialization of the data structure, the data structure invariant needs to have been established. And assuming it is true before an operation of the data structure (and assuming any preconditions of said operation), the data structure invariant is again expected to be true afterwards.

In particular, the *data structure invariants do not have to hold in the middle of the operations on the data structure*. This doesn't happen for the simple bit example, because it has no internal state. But rotations in a binary search tree do not maintain data structure invariants. In fact, they deliberately damage some data structure invariants for the sake of establishing another and then ultimately restore both. Likewise, an array that is automatically sorting itself upon changes will have to violate the sortedness invariant while it is in the process of resorting itself.

The corresponding operational models for the data structure bring to the surface an often overlooked point. The fact that data structure invariants are true *after* an operation *if* they were true before, and the fact that the operational model is a nondeterministic sequential program make it very clear that data structure invariants might fail in concurrent use of data structures by multiple threads at once. Even if the data structure invariant was true when the first thread initiates an operation, it is not yet established if the second thread calls another data structure operation on the same data before the first operation terminates and successfully restores the data structure invariant. Because of that, there is no guarantee that the data structure would hold again even by the time both operations successfully terminated.

Be careful with concurrent uses of data structures that have not been explicitly designed for that purpose!

References

[ABB$^+$16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Matthias Ulbrich, editors. *Deductive Software Verification – The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.

[BP06] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006. doi:10.1007/11814771_23.