03/29/19 Recitation Notes

17-355/17-665/17-819: Program Analysis (Spring 2019)
Jenna Wise[*]
jlwise@andrew.cmu.edu

1 Reminders

- Homework 8 is due next **Tuesday, April 2, 2019 at 11:59pm**. Instructions can be found on the course website

2 Static Verification Tools: Dafny

There are a variety of different static verification tools available for use. Some of the popular ones employing Hoare-style verification are:

- VCC
  - A mechanical verifier for concurrent C programs
- VeriFast
  - A research prototype of a tool for modular formal verification of correctness properties of single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic
- Chalice
  - An experimental language and static verifier that explores specification and verification of concurrency in programs
  - Specifications are written in an Implicit Dynamic Frames logic
- Dafny
  - The rest of this document discusses Dafny in detail

The swMATH website has a more comprehensive and searchable list of static verifiers and their corresponding features.

The rest of this document discusses Dafny, as it is the verification tool that you need to use to complete homework 8.

[*]These recitation notes were developed together with Jonathan Aldrich

2.1 Introduction to Dafny

[1] Dafny is a programming language with built-in specification constructs and static program verifier to verify the functional correctness of programs — making sure the program is doing what the programmer intended it to do.

The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, dynamic allocation, and inductive datatypes, and builds in specification constructs.
The specifications include:

- pre- and postconditions
- assertions
- loop invariants
- frame specifications (read and write sets)
- termination metrics

To further support specifications, the language also offers:

- updatable ghost variables
- recursive functions
- arrays
- quantifiers
- types like sets and sequences

Specifications and ghost constructs are used only during verification; the compiler omits them from the executable code. Section 2.1.2 discusses some of the language features of Dafny in more detail.

The Dafny verifier is run as part of the compiler. As such, a programmer interacts with it much in the same way as with the static type checker — when the tool produces errors, the programmer responds by changing the programs type declarations, specifications, and statements.

The easiest way to try out Dafny is in your web browser at rise4fun. Once you get a bit more serious, you may prefer to download to run it on your machine. Although Dafny can be run from the command line (on Windows or other platforms), the preferred way to run it is in Microsoft Visual Studio 2010, where the Dafny verifier runs in the background while the programmer is editing the program.

The Dafny verifier is powered by Boogie and Z3, as discussed in more detail in Section 2.1.1.

2.1.1 Dafny Verifier Architecture

The Dafny verifier is built on top of Boogie and Z3, as seen in 1

Boogie is an intermediate verification language and verification tool on which to build program verifiers for other languages. As you can see in Figure 1, the VCC and HAVOC verifiers for C and the verifiers for Dafny, Chalice, and Spec# are all built on top of Boogie.

[1]The information in this section comes largely from the following resources: Dafny Online Tutorial, Dafny's Website, Boogie's Website.

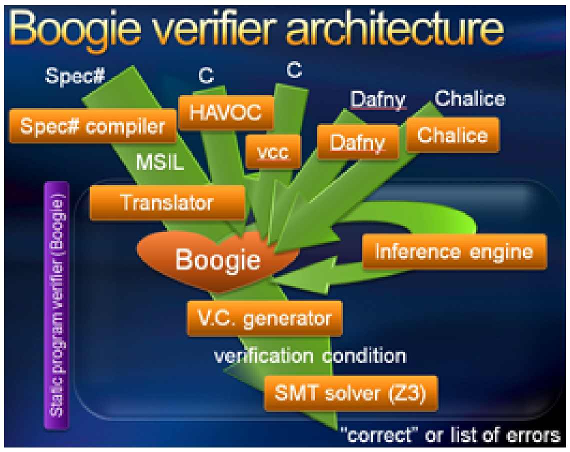

Figure 1: The Dafny/Boogie verifier architecture

The Boogie verifier (verification tool) accepts the Boogie language as input, optionally infers some invariants in the given Boogie program, and then generates verification conditions that are passed to an SMT solver (as seen in Fig. 1). The default SMT solver is Z3 (Z3 was discussed in a previous recitation).

2.1.2 Dafny Specification Basics

This section briefly presents some of the specification language features in Dafny. For more information on these and other features see the Dafny Online Tutorial or Section 2.3.

Pre- and Postconditions:
We use pre- and postconditions in Dafny specify the behavior of methods and prove that the behavior we claim of the method is true.

A postcondition is something that is true after the method returns. Postconditions, declared with the ensures keyword, are given as part of the method's declaration, after the return values (if present) and before the method body. The keyword is followed by a boolean expression: something that can be true or false. When the expression is true, we say that the postcondition holds.

A precondition is similar to a postcondition, except that it is something that must be true before a method is called. When you call a method, it is your job to establish (make true) the preconditions, something Dafny will enforce using a proof. Likewise, when you write a method, you get to assume the preconditions, but you must establish the postconditions. The caller of the method then gets to assume that the postconditions hold after the method returns. Not all methods necessarily have preconditions. Preconditions have their own keyword, requires.

The following example illustrates pre- and postconditions in Dafny,

Listing 1: Pre- and postconditions in Dafny

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
   requires 0 < y
   ensures less < x
   ensures x < more
   // Equivalently:
   // ensures less < x && x < more
   // Or:
```

```
   // ensures less < x < more
   // preconditions have the same equivalent forms
{
   more := x + y;
   less := x - y;
}
```

Assertions:
Unlike pre- and postconditions, an assertion is placed somewhere in the middle of a method. Like the previous two annotations, an assertion has a keyword, assert, followed by the boolean expression and the semicolon that terminates simple statements. An assertion says that a particular expression always holds when control reaches that part of the code. For example, the following is a trivial use of an assertion inside a dummy method:

Listing 2: Assertions in Dafny

```
method Testing()
{
   assert 2 < 3;
}
```

Dafny proves this method correct, as 2 is always less than 3. Asserts have several uses, but chief among them is checking whether your expectations of what is true at various points is actually true. Assertions are a powerful tool for debugging annotations, by checking what Dafny is able to prove about your code.

Loop Invariants:
To make it possible for Dafny to work with loops, you need to provide loop invariants, another kind of annotation.

A loop invariant is an expression that holds upon entering a loop, and after every execution of the loop body. It captures a property that is invariant, i.e. does not change, about every step of the loop. For example, we see in the above loop that if i starts off positive, then it stays positive. So we can add the invariant, using its own keyword, to the loop:

Listing 3: Loop invariants in Dafny

```
   var i := 0;
   while i < n
      invariant 0 <= i
   {
      i := i + 1;
   }
```

When you specify an invariant, Dafny proves two things:

1) the invariant holds upon entering the loop

2) the invariant is preserved by the loop — assuming that the invariant holds at the beginning of the loop, we must show that executing the loop body once makes the invariant hold again

Dafny can only know upon analyzing the loop body what the invariants say, in addition to the loop guard (the loop condition). Just as Dafny will not discover properties of a method on its own, it will not know any but the most basic properties of a loop are preserved unless it is told via an invariant.

In our example, after we exit the loop, we will have that $i == n$, because i will stop being incremented when it reaches n. We can use our assertion trick to check to see if Dafny sees this fact as well:

Listing 4: Loop invariants in Dafny

```
var i: int := 0;
while i < n
   invariant 0 <= i
{
   i := i + 1;
}
assert i == n;
```

We find that this assertion fails. As far as Dafny knows, it is possible that i somehow became much larger than n at some point during the loop. All it knows after the loop exits (i.e. in the code after the loop) is that the loop guard failed, and the invariants hold. In this case, this amounts to $n <= i$ and $0 <= i$. But this is not enough to guarantee that $i == n$, just that $n <= i$. Somehow we need to eliminate the possibility of i exceeding n. One first guess for solving this problem might be the following:

Listing 5: Loop invariants in Dafny

```
var i := 0;
while i < n
   invariant 0 <= i < n
{
   i := i + 1;
}
```

This does not verify, as Dafny complains that the invariant is not preserved (also known as not maintained) by the loop. We want to be able to say that after the loop exits, then all the invariants hold. Our invariant holds for every execution of the loop except for the very last one. Because the loop body is executed only when the loop guard holds, in the last iteration i goes from $n - 1$ to n, but does not increase further, as the loop exits. Thus, we have only omitted exactly one case from our invariant, and repairing it is relatively easy:

Listing 6: Loop invariants in Dafny

```
...
   invariant 0 <= i <= n
...
```

Now we can say both that $n <= i$ from the loop guard and $0 <= i <= n$ from the invariant, which allows Dafny to prove the assertion $i == n$. The challenge in picking loop invariants is finding one that is preserved by the loop, but also that lets you prove what you need after the loop has executed.

Arrays & Quantifiers:

Arrays. Arrays are a built in part of the Dafny language, with their own type, $array < T >$, where T is another type. For now we only consider arrays of integers, $array < int >$. Arrays can be null, and have a built in length field, $a.Length$. Element access uses the standard bracket syntax and are indexed from zero, $a[0]$. All array accesses must be proven to be within bounds, which is part of the no-runtime-errors safety guarantee by Dafny. Because bounds checks are proven at verification time, no runtime checks need to be made. To create a new array, it must be allocated with the new keyword.

One of the most basic things we might want to do with an array is search through it for a particular key, and return the index of a place where we can find the key if it exists. We have two outcomes for a search, with a different correctness condition for each. If the algorithm returns an index (i.e. non-negative integer), then the key should be present at that index. This might be expressed as follows:

Listing 7: Arrays in Dafny

```
method Find(a: array<int>, key: int) returns (index: int)
   ensures 0 <= index ==> index < a.Length && a[index] == key
{
   // Open in editor for a challenge...
}
```

The array index here is safe because the implication operator is short circuiting. Short circuiting means if the left part is false, then the implication is already true regardless of the truth value of the second part, and thus it does not need to be evaluated. Using the short circuiting property of the implication operator, along with the boolean ”and” (&&), which is also short circuiting, is a common Dafny practice. The condition $index < a.Length$ is necessary because otherwise the method could return a large integer which is not an index into the array. Together, the short circuiting behavior means that by the time control reaches the array access, index must be a valid index.

If the key is not in the array, then we would like the method to return a negative number. In this case, we want to say that the key is not in the array. To express this property, we turn to another common Dafny tool: quantifiers.

Quantifiers. A quantifier in Dafny most often takes the form of a forall expression, also called a universal quantifier. As its name suggests, this expression is true if some property holds for all elements of some set. For now, we will consider the set of integers. An example universal quantifier, wrapped in an assertion, is given below:

Listing 8: Quantifiers in Dafny

```
   assert forall k :: 0 <= k < a.Length ==> ...a[k]...;
```

A quantifier introduces a temporary name for each element of the set it is considering. This is called the bound variable, in this case k. The bound variable has a type, which is almost always inferred rather than given explicitly and is usually int anyway. A pair of colons (::) separates the bound variable and its optional type from the quantified property (which must be of type bool).

The example above, says that some property holds for each element of the array. The implication makes sure that k is actually a valid index into the array before evaluating the second part of the expression. Dafny can use this fact not only to prove that the array is accessed safely, but also reduce the set of integers it must consider to only those that are indices into the array.
With a quantifier, saying the key is not in the array is straightforward:

Listing 9: Quantifiers in Dafny

```
   forall k :: 0 <= k < a.Length ==> a[k] != key
```

2.2 The Frame Problem with Dafny

[2] Consider this program written in Dafny:

Listing 10: The frame problem with Dafny

```
class Cell {
    var contents: int;

    method Init()
        ensures contents == 1;
    {
        contents := 1;
    }

    method setContents(x: int)
        ensures contents == x;
    {
        contents := x;
    }
}

method TestCell()
{
    var c1 := new Cell;
    var c2 := new Cell;
    c1.Init();
    c2.Init();

    c1.setContents(4);

    assert c1.contents == 4;
    assert c2.contents == 1;
}
```

This program will not verify correctly in Dafny, because we do not provide specifications that allow us to solve "The Frame Problem". The frame problem can be motivated by this example program. In particular, when the method $TestCell()$ calls $c1.setContents(4)$, the $setContents$ method's specification makes no indication of what else may happen to the program state during the execution of $c1.setContents(4)$ other than assigning $c1.contents$ to 4. Therefore, the specification is too weak to say that $c2.contents$ has not changed and is still equal to 1 after the call to $c1.setContents(4)$.

We can state the frame problem as follows: "When formally describing a change in a system, how do we specify what parts of the state of the system are not affected by that change?"
The frame problem can be solved by using one of the following verification techniques:

- separation logic
- implicit dynamic frames

[2] The information in this section comes largely from the following resources: Dafny Online Tutorial, Dynamic Frames and Automated Verification.

- dynamic frames

Dafny uses dynamic frames to solve the frame problem and is discussed briefly in Section 2.2.1

2.2.1 Framing in Dafny

The idea behind Dynamic Frames, is the introduction of the footprints of method and functions. The footprint of a method is the set of all fields that the method is permitted to modify; and, the footprint of a pure function is the set of all fields that the function is permitted to read.

So, methods (functions) are required to list which parts of memory they modify (read/rely on), with a modifies (reads) annotation. A modifies (reads) annotation is not a boolean expression, like the other annotations we have seen, and can appear anywhere along with the pre- and postconditions. Instead of a property that should be true, it specifies a set of memory locations that the method modifies (function reads). The name of an array, like a in the above example, stands for all the elements of that array. One can also specify object fields and sets of objects. Dafny will check that you do not modify (read) any memory location that is not stated in the modifies (reads) frame. This means that method (function) calls within a method (function) must have modifies (reads) frames that are a subset of the calling method's (function's) modifies (reads) frame.

Note that framing only applies to the heap, or memory accessed through references. Local variables are not stored on the heap, so they cannot be mentioned in modifies annotations. Arrays and objects are reference types, and they are stored on the heap (though as always there is a subtle distinction between the reference itself and the value it points to.)

In combination with reads, modification restrictions allow Dafny to prove properties of code that would otherwise be very difficult or impossible (as seen in Sec. 2.2). Reads and modifies are one of the tools that allow Dafny to work on one method at a time, because they restrict what would otherwise be arbitrary modifications of memory to something that Dafny can reason about. For example, the following example employs dynamic frames annotations in Dafny to reason about framing and prove the Cell program correct:

Listing 11: Framing in Dafny

```
class Cell {
    var contents: int;

    method Init()
        modifies this;
        ensures contents == 1;
    {
        contents := 1;
    }

    method setContents(x: int)
        modifies this;
        ensures contents == x;
    {
        contents := x;
    }
}

method TestCell()
```

```
{
    var c1 := new Cell;
    var c2 := new Cell;
    c1.Init();
    c2.Init();

    c1.setContents(4);

    assert c1.contents == 4;
    assert c2.contents == 1;
}
```

2.3 Helpful Dafny Resources

- Dafny Github Repository
- Dafny's Website
- Dafny Online Tutorial
- Dafny Online