Automated Program Verification and Testing
15414/15614 Fall 2016
Lecture 4:
Introduction to Dafny

Matt Fredrikson
mfredrik@cs.cmu.edu

October 17, 2016
The goal of this lecture is to:
  ▶ Cover enough Dafny to get started on the next assignment.
  ▶ Touch on some important things we’ll cover in detail later.

We won’t cover some of Dafny’s coolest features
  ▶ More on these in future lectures...

Consult references and tutorials provided at the end of these slides.
Dafny is a programming language, verifier, and compiler

Designed from the ground-up with static verification in mind

Uses SMT solver to automatically prove correctness

When this is not possible, requests proof annotations
Dafny makes it easier to write *correct* code

Correctness means two things:
- No runtime errors: null deref., div. by 0, index o.o.b, ...
- Program does what you intended
- Terminates (when applicable)

Your intentions are captured with a *specification*
Can’t I still write the wrong specification?

Specifications should be:

- High-level expression of the desired behavior
- Shorter and more direct than implementation
- Not concerned with efficiency and representation

\[
\text{forall } k : \text{int} :: 0 \leq k < a.\text{Length} \implies 0 < a[k]
\]

\[
\text{exists } I : \text{Interpretation} :: \text{fmla_satisfied}(F, I)
\]
Specifications in Dafny can be arbitrarily sophisticated

Effectively, Dafny can be seen as hosting two sub-languages

1. Imperative, executable core: methods, loops, arrays, if statements...

2. Functional specification language: pure functions, sets, predicates, algebraic datatypes, “ghost” state, ...

The code you write to specify and prove things is not compiled
The Tool

Dafny Compiler

C# Code

.NET Compiler

.NET Executable

Error Messages

Dafny Verifier

Boogie

SMT Solver (Z3)
Dafny Basics: Methods

Unit of executable code

Note that:

▶ Types for parameters and return values are required
▶ Types are given after names, followed by “:”
▶ Return values are named

Methods can have multiple return values

```dafx
method Abs(x: int) returns (r: int)
{
    ...;
}

method M() returns (r1: int, r2: int)
{
    ...;
}
```
Dafny Basics: Methods

```daffny
method MultipleReturns(x: int, y: int) returns (r1: int, r2:int)
{
    r1 := x + y;
    r2 := x - y;
    // Comments are given
    /* in typical C/Java fashion */
}
```

To return a value, assign to the named return variable

You can assign to the same return value multiple times

Assignments use :=, not =

No valid syntax in Dafny uses a single =
You can also use return statements

Input parameters are always read-only

Compound statements (if, while, ...) always need curly braces
method MoreOrLess(x: int, y: int) returns (more: int, less: int)
   ensures less < x
   ensures x < more
{
    more := x + y;
    less := x - y;
}

Expression that is always true after method executes

These are statically-checked by Dafny

Note: could have also written less < x < more

Will Dafny accept these postconditions?
method MoreOrLess(x: int, y: int) returns (more: int, less: int)
    requires 0 < y
    ensures less < x < more
{
    more := x + y;
    less := x - y;
}

Expression that must be true when method is called

Again, these are statically-checked by Dafny

Your job: assume pre-conditions, make sure post-conditions hold
method TestCase(x: int)
{
    var v := Abs(x);
    assert 0 <= v;
}

Expression that must be true when execution reaches statement

Dafny will attempt to prove that the assertion holds

Aside: local variable types can usually be inferred

Notice: methods can’t be called from Boolean exprs.

Why?
method Abs(x: int) returns (y: int)
  ensures 0 <= y
{
  if (x < 0) { return -x; }
  else { return x; }
}

method TestCase()
{
  var v := Abs(3);
  assert v == 3;
}

Dafny won’t be able to prove this

Forgets everything about other methods

...except what the postconditions say

This is crucial for making verification feasible!

When using methods:

▶ Assume pre- and post-conditions describe them entirely
▶ They could be any method that satisfies the specification
method Abs(x: int)
  returns (y: int)
  ensures 0 <= y
{
  y := 0
}

Satisfies the specification...

Perfect!

...but redundant?
Dafny: Functions

```dafny
define function abs(x: int): int
{ if x < 0 then -x else x }
```

Dafny doesn't forget about function bodies

```dafny
assert abs(3) == 3;
```

Think: pure mathematical functions

- Cannot write to memory
- Body is a single expression
- Single return value
- Not compiled and executed

Used directly in annotations

- Pre-, post-conditions
- Assertions
- Invariants
Loops present a problem: how many times will it execute?

Invariants let Dafny make assumptions about what carries through any number of loop executions.

```
var i := 0;
while (i < n)
    invariant 0 <= i;
{
    i := i + 1;
}
```
method ComputeFib(n: nat)
    returns (b: nat)
    ensures b == fib(n);
{
    if (n == 0) { return 0; }
    var i := 1;
    var a := 0;
    b := 1;
    while (i < n)
        invariant 0 < i <= n;
        invariant a == fib(i - 1);
        invariant b == fib(i);
        { 
        a, b := b, a + b;
        i := i + 1;
        }
}

function fib(n: nat): nat
{
    if n == 0 then 0 else
    if n == 1 then 1 else
        fib(n - 1) + fib(n - 2)
    }

As with methods, Dafny forgets everything about loop bodies

Use the loop guard + invariants to establish a fact after the loop terminates
Dafny: Termination

Dafny proves termination

Obviously, you need to help

Specification element: decreases annotation
  ▶ Attach to loops and recursive functions
  ▶ Provide termination metric

Termination metric:
  ▶ Gets smaller every iteration
  ▶ Has a lower bound

```
while (i < n)
    invariant 0 <= i <= n;
    decreases n - i;
{
    i := i + 1;
}
```

**Note:** tell Dafny not to prove termination by specifying:

```
decreases *
```
Arrays are built into the language

- They have type `array<T>`
- Can be null
- Have built-in `Length` field
- Initialized with `new`
- Accessed with `[ brackets ]`

Dafny checks bounds statically

```dafny
method M(x: int)
{
    var a := new int[10];
    var b := a[x];  // ERROR
    if (0 <= x < 10) {
        b := a[x];  // OK
    }
}

method M(x: int, c: array<int>)
    requires 0 <= x < 10
{
    var a := new int[10];
    var b := a[x];  // OK
    if (0 <= x < c.Length) {
        // ERROR
    }
}
```
Shared memory makes verification hard

Dafny uses framing annotations to specify:

- which regions of memory a function can read (“read frame”)
- and which regions methods can modify (“write frame”)

```c
function f(a: array<int>): int
    reads a
    { 
      sum(a) + prod(a) 
    }

method M(a: array<int>,
    b: array<int>)
    modifies a
    { 
      if(a != null && b != null) {
        b[0] := a[0];  // ERROR
      }
    }
```
Inductive datatypes are created using a set of *constructors*

For each constructor $C_t$, Boolean field $C_t$?

Can also match using `match` statement

```daffny
datatype Tree =
  Empty
  | Node(l: Tree, d: int, r: Tree);
...
if (t.Empty?) { ... }
else if (t.Node?) {
  d := t.data;
}
```

```daffny
match(t) {
  case Empty => ...
  case Node(l, d, r) =>
    ...
}
```
var g: seq<int> := [];
g := g + [0, 1, 2];
assert |g| == 3;
assert g[0..1] == [0, 1];
assert g[2] == 2;
assert g[..] == [0, 1, 2];
assert 0 in h;
assert 3 !in h;

Immutable type: cannot be modified once created

No need to allocate: sequences are values

Ordered list of values

Used in both specification and code
**Strongly encouraged:** complete the main tutorial at

http://rise4fun.com/Dafny/tutorial

Getting started guide: http://goo.gl/mJ1Grr

Slightly older guide: http://goo.gl/MVYsbq

Main webpage: http://goo.gl/G1XDik

Reference manual: http://goo.gl/IGVbYY (note: this is a work in progress)
Second assignment goes out later today

Main task: implement a SAT solver

Requires the ability to compile Dafny on your machine

Get started early!