Dafny

- A Language and Program Verifier for Functional Correctness
- Designed to support static verification of programs
  - https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/
  - https://en.wikipedia.org/wiki/Dafny
  - https://dafny.org/dafny/
- Available for download
  - https://github.com/Microsoft/dafny
- Getting Started with Dafny: A Guide
  - https://dafny.org/dafny/OnlineTutorial/guide
  - https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml220.pdf
- Dafny Reference Manual
  - https://dafny.org/dafny/DafnyRef/DafnyRef
  - https://dafny.org/dafny/DafnyRef/out/DafnyRef.pdf
- Dafny Cheatsheet
  - https://docs.google.com/document/d/1kz5_yqzhrEyXII96eCF1YoHZhnb_6dzv-K3u79bMMis/edit?pref=2&pli=1

Language Features

- Dafny is an imperative, class-based language
- Methods
- Variables
- Types
- Type parameters (“Generics”)
- Loops
- If statements
- Arrays
- No support for subclasses and no constructors
- Influenced by Java and C#
- No-runtime-errors safety guarantee

Dafny uses annotations to reason about code

- Generates a proof that the code matches the annotations
- Annotations are a form of specification
- Example `forall k: int :: 0 <= k < a.Length ==> 0 < a[k]`
  - All elements of array a are greater than 0.
- Proves that there are no runtime errors, null references, etc.
- Syntax is unique
  - Not the same as Java, C++, etc.
  - Targets C#

Dafny Basics

- the smallest unit of verification is the method
- may have multiple returns
- assignment operator is :=
- preconditions use the requires keyword
- postconditions use the ensures keyword

```
method MethodName(x: int, y: int) returns (z: int, w: int)
    requires x == 0 && y >= 0    // PRECONDITION
    ensures z != 0 || w != 0    // POSTCONDITION
{

  ...

}
```

Data types

- Value types: those whose values do not lie in the program heap
  - Scalar
    - bool
    - char
    - int
    - nat
    - real
  - Collections
    - set
    - multiset
    - seq
    - string
    - map

Data types

- Reference types: represent *references* to objects allocated dynamically in the program heap
  - Class types
  - Traits
  - Array types
  - Can be *nullable* (contain the special `null` value) *and non-null types*

Sequences

- an immutable (cannot be modified once created) mathematical list
- can be used to represent many ordered collections, including lists, queues, stacks, etc.
- no reads clause is necessary
- seq<T>
- length of a sequence is written |s|

```
predicate sorted2(s: seq<int>)
{
  0 < |s| ==> (forall i :: 0 < i < |s| ==> s[0] <= s[i]) && sorted2(s[1..])
}

method m()
{
  var s := [1, 2, 3, 4, 5];
  assert s[|s|-1] == 5; //access the last element
  assert s[|s|-1..|s|] == [5]; //slice just the last element, as a singleton
  assert s[1..] == [2, 3, 4, 5]; // everything but the first
  assert s[..|s|-1] == [1, 2, 3, 4]; // everything but the last
  assert s == s[0..] == s[..|s|] == s[0..|s|]; // the whole sequence
}
```

Arrays

- mutable heap allocated (potentially aliased)
- accessed by pointers
- must be allocated with the `new` keyword
- array<T> (array?<T> is possibly-null arrays)
- have a built-in length field, a.Length

```
method m()
{
  var a := new int[][42,43,44]; // 3 element array of ints
  a[0], a[1], a[2] := 0, 3, -1;
  assert a[1..] == [3, -1];
  assert a[..1] == [0];
  assert a[1..2] == [3];
}
```

Boolean operations

Operator	Precedence	Description
<==>	1	equivalence (if and only if)
==>	2	implication (implies)
<==	2	reverse implication (follows from)
&&	3	conjunction (and)
||	3	disjunction (or)
==	4	equality
!=	4	disequality
!	10	negation (not)

Short circuiting!!!

Methods and functions

- Method
  - a piece of imperative, executable code
  - body consists of a series of statements
  - may have local variables are declared with the var keyword
  - Dafny “forgets” about the body of every method except the one it is currently working on
- Ghost function
  - body must consist of exactly one expression, with the correct type
  - can be used directly in specifications
  - can only appear in annotations
  - never part of the final compiled program, just a tool to help us verify our code
  - Dafny does not forget the body of a function when considering other functions
- Function (function method)
  - can be called from real code

Predicates

- Predicate
  - a function which returns a Boolean
  - no return type, because predicates always return a Boolean

```
predicate sorted(a: array<int>)
  reads a
{
  forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
```

Dafny Examples

- Hello World in Dafny

```
method Main() {
  print "hello, Dafny\n";
  assert 10 < 2;   // this assertion fails
}
```

- Fibonacci

```
function Fibonacci(n: int): int
  decreases n   // this recursive condition is violated
{
  // what is wrong here?
  if n < 2 then n else Fibonacci(n+2) + Fibonacci(n+1)
}
```

This should be

```
function Fibonacci(n: int): int
  decreases n
{
  if n < 2 then n else Fibonacci(n-2) + Fibonacci(n-1)
}

// Decreases is like a decrementing function
```

Dafny Basics

- Assertions: placed somewhere in the middle of a method

```
method  Abs(x: int) returns (x': int)
{
  x' := x;
  if(x' < 0) { x' := x' *  -1; }
}

method Testing()
{
   var v := Abs(3);
   assert v == 3;
   assert 0 < v;
   assert 0 <= v;
}
```

```
method  Abs(x: int) returns (x': int)
  ensures x' >= 0
  ensures (x < 0 && x' ==  -1*x) || (x' == x)
{
  x' := x;
  if(x' < 0) { x' := x' *  -1; }
}

method Testing()
{
   var v := Abs(3);
   assert v == 3;
   assert 0 < v;
   assert 0 <= v;
}
```

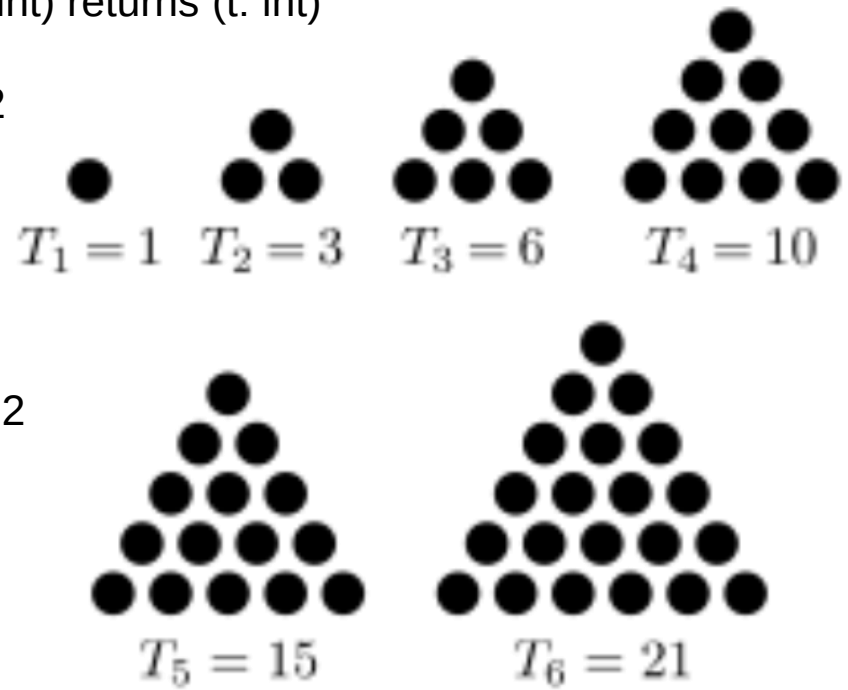
```
function abs(x: int): int
{
  if x < 0 then -x else x
}

method m()
{
  assert abs(3) == 3;
}
```

Dafny Examples

```
method TriangleNumber(N: int) returns (t: int)
  requires N >= 0
  ensures t == N * (N + 1) / 2
{
  t := 0;
  var n := 0;
  while n < N
    invariant 0 <= n < N
    invariant t == n * (n + 1) / 2
  {
    n, t := n + 1, t + n + 1;
  }
}
```

```
method TriangleNumber(N: int) returns (t: int)
  requires N >= 0
  ensures t == N * (N + 1) / 2
{
  t := 0;
  var n := 0;
  while n < N
    invariant 0 <= n <= N
    invariant t == n * (n + 1) / 2
    decreases N – n
  {
    n, t := n + 1, t + n + 1;
  }
}
```



Based on Fig. 1, Developing Verified Programs with Dafny by Rustan Leino

Dafny Examples

- Compute x + y

```
method Add(x: int, y: int) returns (r: int)
  requires 0 <= x && 0 <= y  // either the postcondition or precondition is violated
  ensures r == 2*x + y          // change to ensures r = x + y
{
  r := x;
  var n := y;
  while n != 0
    invariant r == x+y-n && 0 <= n  // loop invariant
  {
    r := r + 1;
    n := n - 1;
  }
}
```

Dafny Examples

- Recursively multiply x * y

```
method Mul(x: int, y: int) returns (r: int)
  requires 0 <= x && 0 <= y
  ensures r == x*y
  decreases x
{
  if x == 0 {
    r := 0;
  } else {
    var m := Mul(x-1, y);  // var declares a new variable
    r := m + x;                  // is this correct? Should be r = m+y?
  }
}
```

Dafny Examples

```
// Can you make the program verify?
method M(n: int) returns (r: int)
  ensures r == n
  // what precondition do we need?
{
  var i := 0;
  while i < n
   // what invariant do we need here?
  {
    i := i + 1;
  }
  r := i;
}
```

Dafny Examples

Needs requires and ensures
Needs a break; statement after leap year test
Loop needs a decreases statement

```
// a function returning a bool
predicate method isLeapYear(y: int) {
  y % 4 == 0 && (y % 100 != 0 || y % 400 == 0)
}

// Does this method terminate?
method WhichYear_InfiniteLoop(d: int) returns (year: int) {
  var days := d;
  year := 1980;
  while days > 365 {
    if isLeapYear(year) {
      if days > 366 {
        days := days - 366;
        year := year + 1;
      }
    } else {
      days := days - 365;
      year := year + 1;
    }
  }
}
```

Solution for the preceeding slide

```
method WhichYear_InfiniteLoop(d: int) returns (year: int)
requires d > 0
ensures year >= 1980
{
 var days := d;
 year := 1980;
 while days > 365
 decreases days
 {
  if isLeapYear(year) {
   if days > 366 {
    days := days - 366;
    year := year + 1;
   }
   else {
    break;
   }
  } else {
   days := days - 365;
   year := year + 1;
  }
 }
}
```

there is an infinite loop if it’s a leap year and days is equal to 366

Dafny Examples

```
method Find(a: array<int>, key: int) returns (index: int)
  requires a != null
  ensures 0 <= index ==> index < a.Length && a[index] == key
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
  index := 0;
  while index < a.Length
    invariant 0 <= index <= a.Length
    invariant forall k :: 0 <= k < index ==> a[k] != key
  {
    if a[index] == key { return; }
    index := index + 1;
  }
  index := -1;
}
```

Binary Search

```
predicate sorted(a: array<int>)
  requires a != null
  reads a
{
  forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
method BinarySearch(a: array<int>, value: int) returns (index: int)
  requires a != null && 0 <= a.Length && sorted(a)
  ensures 0 <= index ==> index < a.Length && a[index] == value
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
  var low, high := 0, a.Length;
  while low < high
    invariant 0 <= low <= high <= a.Length
    invariant forall i ::
      0 <= i < a.Length && !(low <= i < high) ==> a[i] != value
  {
    var mid := (low + high) / 2;
    if a[mid] < value {
      low := mid + 1;
    }
    else if value < a[mid] {
      high := mid;
    }
    else {
      return mid;
    }
  }
  return -1;
}
```

Debugging

- Show counterexample (F7 in VSCode)
- Main method
- /compile:4
- print statements, e.g.:

```
print "power(", m, ", ", n, ") = ", c, "\n";
```