Practical Formal Methods

Specifying and Verifying Programs in Dafny

Examples: https://github.com/formal-and-practical/dafny-examples

Program Correctness

Is this program fragment correct?

```
x = 0;
y = a;
while (y > 0) {
  x = x + b;
  y = y - 1;
}
```

Recall: A program can only be said to be correct
with respect to a specification

Correctness

Is this program fragment correct with respect to the following specification?

“Given integers a and b, the program produces in x the product of a and b”

```
x = 0;
y = a;
while (y > 0) {
  x = x + b;
  y = y - 1;
}
```

Correctness

Is this program fragment correct with respect to the following specification?

*"Given **positive** integers a and b,*
the program produces in x the product of a and b"

```
x = 0;
y = a;
while (y > 0) {
  x = x + b;
  y = y - 1;
}
```

Design by Contract

Specification of example program:

"Given positive integers a and b,
the program produces in x the product of a and b"

requires a and b to be positive integers

ensures x is the product of a and b

Precondition: caller needs to ensure this to get a meaningful result

Postcondition: callee guarantees this when precondition is met

Timsort

- Timsort is a sorting algorithm developed for Python by Tim Peters in 2002.
- It uses a combination of merge sort and insertion sort.
- It was designed to perform well on real-world data (with *runs* of descending values, and of non-descending values).
- Ported to Java 1.7 (`java.util.Collections.sort` and `java.util.Arrays.sort`) in 2011.
- Default sorting algorithm for Android SDK, Oracle's JDK and Open JDK.

Timsort bug

Bug in Timsort discovered in 2013, fixed in Java SE 9

```
git clone git@github.com:abstools/java-timsort-bug.git
cd java-timsort-bug
javac *.java
java TestTimSort 67108864
```

used to lead to

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 40
at java.util.TimSort.pushRun(TimSort.java:413)
at java.util.TimSort.sort(TimSort.java:240)
at java.util.Arrays.sort(Arrays.java:1438)
at TestTimSort.main(TestTimSort.java:18)
```

Stijn de Gouw
CWI, The Netherlands

Formal verification

To formally verify a computer program you need

- A formal (i.e., mathematical) specification
- A formal proof
- Automated tools (Timsort found using the KeY tool)
- Expertise

Learning about specification and proof sharpens thinking

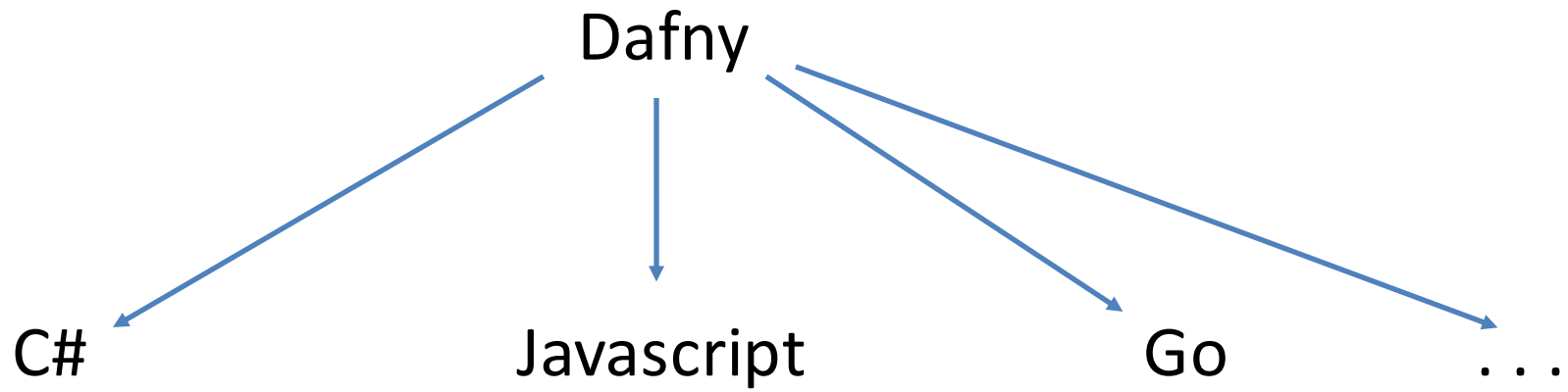
Formal verification

Some program verification tools

- KeY, OpenJML – Java
- VCC, Verifast, Smack – C
- Spec# – C#
- Stainless, Sireum – Scala

- Why3 – WhyML
- Dafny – Dafny

Formal verification



Educational objectives

Learn how to

- specify precisely what a program is supposed to do
- verify that a program behaves as specified
- derive a program that behaves as specified
- use the Dafny programming language and verifier for that

Introduction to Dafny

```
method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  var y := 2 * x;
  r := x + y;
}
```

The caller should not be able to see a method's body, only its specification

The specification describes the method's behavior, abstracting from the details of the method's body

Introduction to Dafny

```
method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  var y := Double(x);
  r := x + y;
}

method Double(x: int) returns (r: int)
  ensures r == 2 * x
```

Introduction to Dafny

```
method Triple(x: int) returns (r: int)
  requires x >= 0
  ensures r == 3 * x
{
  var y := Double(x);
  r := x + y;
}

method Double(x: int) returns (r: int)
  requires x >= 0
  ensures r == 2 * x
```

Introduction to Dafny

```
method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  if x >= 0 {
    var y := Double(x); r := x + y;
  } else {
    var y := Double(-x); r := x - y;
  }
}

method Double(x: int) returns (r: int)
  requires x >= 0
  ensures r == 2 * x
```

Logic in Dafny

```
true false
!A                          "not A"
A && B                      "A and B"
A || B                      "A or B"

A ==> B                     "A implies B"   or   "A only if B"
A <==> B                    "A if and only if B"

Precedence order:   !     &&     ||     ==>     <==>

forall x :: A               "for all x, A is true"
exists x :: A               "there exists an x such that A is true"
```

Program state

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  var a := x + 3;
  var b := 12;
  y := a + b;
}
```

The program variables x, y, a, and b, collectively constitute the method's *state*

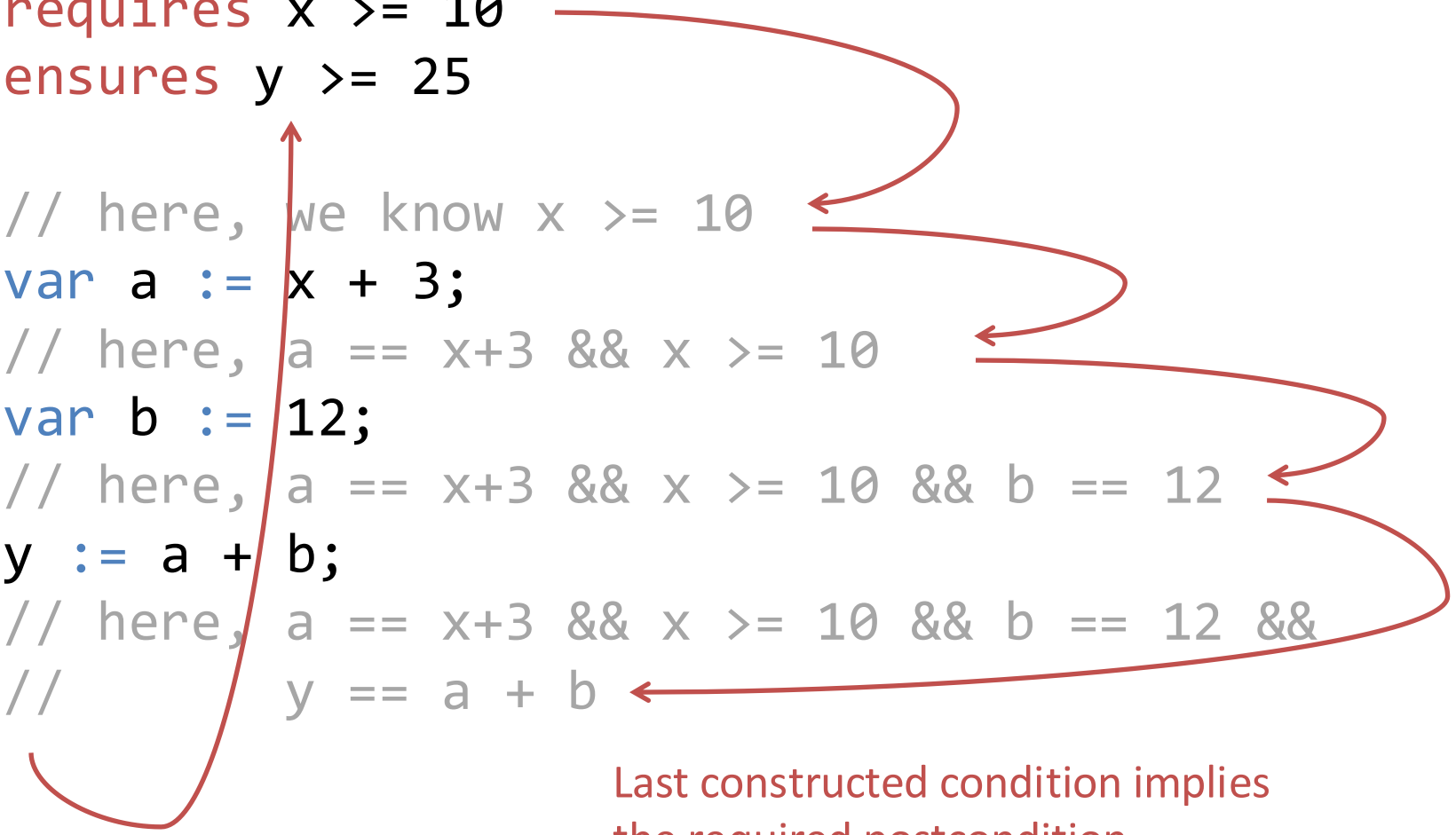
Note: not all program variables are in scope the whole time

Floyd-Hoare logic (forward reasoning)

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we know x >= 10
  var a := x + 3;
  // here, a == x+3 && x >= 10
  var b := 12;
  // here, a == x+3 && x >= 10 && b == 12
  y := a + b;
  // here, a == x+3 && x >= 10 && b == 12 &&
  //       y == a + b
}
```

Floyd-Hoare logic (forward reasoning)

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we know x >= 10
  var a := x + 3;
  // here, a == x+3 && x >= 10
  var b := 12;
  // here, a == x+3 && x >= 10 && b == 12
  y := a + b;
  // here, a == x+3 && x >= 10 && b == 12 &&
  //       y == a + b
}
```



Last constructed condition implies
the required postcondition

Floyd-Hoare logic (weakest preconditions)

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we want  x + 3 + 12 >= 25
  var a := x + 3;
  // here, we want  a + 12 >= 25
  var b := 12;
  // here, we want  a + b >= 25
  y := a + b;
  // here, we want  y >= 25
}
```

Floyd-Hoare logic (weakest preconditions)

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we want  x + 3 + 12 >= 25
  var a := x + 3;
  // here, we want  a + 12 >= 25
  var b := 12;
  // here, we want  a + b >= 25
  y := a + b;
  // here, we want  y >= 25
}
```

Last calculated condition is implied by the stated precondition

Exercise 1

Consider a method with the type signature below
that returns s to be the sum of x and y
and m being the maximum of x and y:

```
method MaxSum(x: int, y: int) returns (s: int, m: int)
```

Suggest the postcondition specification (i.e., ensures) for this method

Exercise 2

Consider a method that attempts to reconstruct the arguments x and y from the return values of `MaxSum` in Exercise 1.
In other words, consider a method with the following type signature and same postcondition as the method of Exercise 1:

```
method ReconstructFromMaxSum(s: int, m: int)
returns (x: int, y: int)
```

1. This method cannot be implemented in general (Why?).
2. Suggest a precondition for the method that allows you to implement it.