Handout D: Value Equality

Christoph Reichenbach

November 15, 2019

Comparing two values for equality is an essential operation in many programming languages. However, it is not always obvious what it means for two values to be equal. Most languages agree that $0 = 0$ and $0 \neq 1$. But what about $\frac{1}{2} = \frac{2}{4}$? Here we must distinguish between the intended meaning of a value and its representation.

In the following, we will look at some of the most common forms of equality.

1 Primitive (Structural) Equality

When we compare two integer values, such as 0 and 1, we can directly compare the (short) bit patterns that represent these values in memory. This is efficient, precise, and unambiguous — but unfortunately this technique only works for fixed-size integers, booleans, enumerations, and characters.

2 Floating-Point Equality

Floating point numbers, like integers, are represented as compact bit patterns in memory. In principle we can compare them directly for equality, and many languages permit that. However, arithmetic operations on floating-point numbers are inherently imprecise. Consider the following Python example:

```
>>> 1.1 + 0.1 == 1.2
False
>>> 1.1 + 0.1
1.2000000000000002
```

Here, the addition of 1.1 to 0.1 produced a result that is slightly different from one that we expect arithmetically. This is because of the trade-off that floating point numbers make: they sacrifice precision (and hence robustness) for execution speed.

As a consequence, the programming language Standard ML forbids equality comparisons between floating point numbers. Programmers must instead use less-than and greater-than comparisons, forcing them to think about suitable epsilon environments in which their conditions might hold.

3 Reference Equality

Let us consider an example from the programming language Java:

```
String s = "hello";
return s == "hello";
```

This computation may return false. This is due to how Java (and other programming languages) represent strings: since Java strings may be (almost) arbitrarily long and thus take up arbitrary amounts of memory, Java cannot store them as efficiently as a 32-bit number and must instead place the strings somewhere in memory. Once the strings are in memory, Java represents them with the memory address that contains the beginning of string.

When Java's equality comparison operator ==, compares strings, it only compares the strings' addresses.

However, it is perfectly possible to represent the same string at different memory addresses, and this happens frequently in Java code. Thus, comparing strings for equality with the == operator is a common bug pattern in Java programs.

4 Programmer-defined Equality

To compare the *contents* of two strings, Java programmers must instead use

```
return s.equals("foo");
```

This mechanism hooks into techniques that we will discuss later; its essence is that it relies on *programmer-definable notions of equality*. In Java, the default mode of comparing strings compares them for equality character-by-character. Alternative forms of equality could be to compare strings irrespective of capitalisation, such that "foo" and "FOO" are equal.

5 Structural Equality

Some languages also support a form of equality checking that works analogously to primitive equality, but generalises its idea and scales to objects of arbitrary size. For example, in the language Go, we can define and compare records like the record person in the program below:

```
type person struct {
    name string
    age  int
}

func main() {
        a := person{"A", 20};
        b := person{"A", 20};
        fmt.Println(a == b);
}
```

This program will print out true, since Go here performs structural equality checking. If we instead write

```
a := person{"A", 21};
```

or

```
a := person{"B", 20};
```

the program will print out false, since it considers records to be equal if and only if all fields are equal.

6 Outlook

Most languages use a combination of these equality methods. For example, Python provides two equality checking operators:

- 'is' performs primitive equality checks on built-in types, such as integers, and reference equality checks on other types, while
- '==' is a user-definable equality checking mechanism that defaults to reference equality checking, but for most of Python's built-in types (lists, tuples, etc.) does structural equality checking.

7 Summary

We have discussed the following forms of equality:

- **Primitive Equality**, which directly compares two values by comparing their bit patterns in memory,
- **Structural Equality**, which performs the same kind of comparison, but recursively within potentially more complex structures,
- **Reference Equality**, which checks whether two pointers/references point to the same address in memory,
- **User-Defined Equality**, which can perform arbitrary equality checking but isn't provided by the language itself.