Concepts of Programming Languages
Lecture Notes: Statically and Dynamically Typed Languages

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Safety

Last lecture, we looked at safety in terms of undefined behavior (remember that C has lots of undefined behavior). In this lecture, we discuss type safety. C is a weakly statically typed language, meaning that types are used to define how data is layed out in memory and to compute offsets. In the example below we allocate memory on the heap in order to store integer values (for this, types indicate how much contiguous memory is needed) and we get a pointer to the beginning of that memory location. We can then use the pointer to access data at that location and the type of the pointer indicates what values we can store there and how it is to be interpreted when being read.

```
1  int main() {
2      int *p = (int*) malloc (sizeof(int));
3      *p = 2123456789;
4
5      printf ("(float)*p = %f\n", (float)*p);    /* loss of
           precision */
6      printf ("*(float*)p = %f\n", *(float*)p); /* rubbish */
7
8      int i = 2;
9      char s[] = "three";
10
11     printf ("i*s = %ld\n", i*(long)s);
12 }
```

Since C is weakly statically typed, however, it is not mandatory to stick to this initial interpretation. We can use type casts to reinterpret data in memory, but the results are usually not meaningful. For example, we can interpret the integer value as a float, but will lose precision because floats represent a smaller maximal magnitude as integers since they use some of their memory to represent the decimal number. Even worse, because the type also determines how the data is arranged in memory, when we reinterpret the pointer itself as a pointer to float, we get complete rubbish.

The program above performs unsafe memory access, which occurs whenever a memory location contains data at a given type but being read without permission or interpreted at an incompatible type. Safe languages prevent unsafe memory access by checking types; Scheme, for instance, throws an exception when data is not interpreted according to its type.

Another common example of unsafe memory access in C is when array bounds are not respected. As with simple types, arrays specify contiguous memory whose cells are interpreted

according to a certain type, while the array index advances by the size of that type. Accessing data outside the array bounds overwrites other memory locations.

```
1 int main () {
2     float f = 10;
3     int a[] = { 10 };
4     short i = 10;
5                                 printf ("f=%f, a[0]=%d i=%d\n", f, a[0],
                                      i);
6     a[-1] = 2123456789; printf ("f=%f, a[0]=%d i=%d\n", f, a[0],
          i);
7     a[1]  = 2123456789; printf ("f=%f, a[0]=%d i=%d\n", f, a[0],
          i);
8 }
```

A third common mistake is pointer aliasing on the stack or heap, or misusing pointer arithmetic to modify function pointers.

```
1 int main() {
2     int x = 2123456789;
3     double y = x;
4     printf ("x=%d, y=%f\n", x, y);
5     double *p = &x;
6     double z = *p;
7     printf ("x=%d, z=%f\n", x, z);
8 }
```

Unsafe memory accesses are not only bugs at runtime that produce unintended results, they cause security problems! In order to avoid these issues, safe languages restrict the way data can be accessed. Java, for example,

- disallows pointers to the stack,
- disallows pointer arithmetic,
- disallows explicit deallocation of memory (uses garbage collection)
- checks array bounds
- checks potentially unsafe casts

In summary, traditional systems languages, such as Assembly, C, C++, are purposefully unsafe in order to allow flexibility in accessing memory and perform operations fast, while recent application languages, such as Java, Scheme, Python, are meant to be safe. Recent systems languages, such as Rust and Go, attempt to isolate the unsafe bits of the language. Even in safe languages, however, dynamic checks can allow program flaws go into production, as we will see in the next section. Tony Hoare considers his introduction of `null` into ALGOL a "billion dollar mistake", since he opted for the easy to implement option (null) over other alternatives (such as Option types) and many programming languages since followed this example.

Dynamic and Static Types

For execution purposes, types in a programming language define offsets in memory. For program maintainability, they provide documentation and user-friendly names about those off-

sets. For example, the code snippet below shows a **struct** that combines two elements. The layout in memory of that struct reserves the size of an **int** followed by the size of a pointer to an element of the struct, which means the pointer is offset in memory from the beginning of the **struct** by the size of an integer. Instead of working with the offsets verbatim, we use the names value and next that refer to those offsets.

```
1 struct Node { int value; Node* next; }
2 Node* getNext (Node* x) {
3     return x->next;
4 }
```

On a very low level, the data in memory is without meaning to a compuer and it will gladly perform any operation that we ask. In order to produce meaningful results, types also document and determine what are valid operations that can be performed on the data in memory. For example, subtracting two numbers is a meaningful operation, while subtracting two strings might not be. Since both are just represented as sequences of bits in memory, in order to distinguish between the two we use types that tell us that subtraction is an operation on integers (and floats etc.), but not on strings (or if it is defined on strings, that perhaps it defines removing a suffix from a string).

There are two major approaches for tracking types:

- *Statically typed languages* track types with the compiler; they detect type errors early at compile time, but the downside is that compilers must be conservative and may disallow some uses that are perfectly fine at runtime
- *Dynamically typed languages* store types with objects in memory; they detect type errors late at runtime, but do not need to be conservative in type checking

C is a weakly statically typed language: types are purely used for computing offsets, but C does not enforce types and we can use the cast operator to freely reinterpret data in memory.

Scheme is a dynamically typed language, where dynamic type checking at runtime detects failures.

```
1 #;> (- 5 "hello")
2 Error in -: expected type number, got '"hello"'.
```

When we try to apply the numeric subtraction operator to a number and a string (who, unlike in C, carry their type information at runtime), dynamic type checking detects a failure and refuses to perform the operation.

Example 1 (Statically or dynamically typed?). How can we find out whether a language is statically or dynamically typed? We can ask ourselves whether the type checker is invoked before execution starts. In order to do that, let's defer computation to the function body.

```
1 #;> (define (f) (- 5 "hello"))
2 #;> (f)
3 Error in -: expected type number, got '"hello"'.
```

When we define the function, we notice that it is perfectly acceptable in Scheme, no type error is raised. Only when we actually execute the function, we see the type error. We conclude that there is no type checking before execution, hence the language is dynamically typed rather than statically typed.

Java is a strongly statically typed language: its compiler enforces types during compilation.

```
class Typing01 {
    public static void main (String[] args) {
        int a = 5;
        String b = "hello";
        System.out.println ("Result = " + (a - b));
    }
}
```

Java comes with a cast operator, which lets us defer type checking to runtime; we can manually make any error dynamic by casting! Since Java is strongly typed, objects at runtime carry their type information so that casting is checked at runtime and wrong casts result in runtime type errors. In the example below, we get a `ClassCastException` at runtime telling us about the inappropriate cast from `String` to `Integer`.

```
class Typing01 {
    public static void main (String[] args) {
        int a = 5;
        String b = "hello";
        System.out.println ("Result = " + (a - (int)(Object)b));
    }
}
```

In static languages, not only the objects at runtime have types, the variables that we declare have types. The example below results in a type error telling us that `String` cannot be converted to **int**.

```
class Typing06 {
    public static void main (String[] args) {
        int a = 5;
        a = "hello";
    }
}
```

Using casting, we can convert any static error into a dynamic type error.

```
class Typing06 {
  public static void main (String[] args) {
    int a = 5;
    a = (int)(Object)"hello";
  }
}
```

Now is it necessary to always annotate all types to all variables? Starting from literals, we can follow operators and functions applied to them in order to find out types. This process is called *type inference* and supported by recent editions of Java.

```
class Typing06 {
  public static void main (String[] args) {
    var a = 5;
    a = "hello";
  }
}
```

In dynamic languages, variables do not have types, only values have types. In the example below, expressed in Scheme, we can define a variable to have the value 5 and then change its value subsequently to a string.

```
#;> (define (main)
      (define a 5)
      (set! a "hello")
      (display a)
  )
#;> (main)
"hello"
```

The tradeoffs of dynamic vs. static typing are summarized below.

- Dynamic languages are more flexible, usually conceptually simpler, compile faster, and it is easier to generate, run, and modify code at runtime
- Static languages detect type errors early at compile time, do not need unit tests for type checking, have automatic (basic) documentation, are faster at runtime because types do not need to be checked and code can be optimized, and consume less memory at runtime

Static types, however, are conservative.

```
int f(int i, String s) {
  return true ? i : s;
}
```

Since in a statically typed language there is a type hierarchy prescripted at compile time, the inferred type for the code snippet above is Object, which is the closest common ancestor to both **int** and String in Java's type hierarchy. In a dynamically typed language, the type of function f would be either **int** or String (we will see Either types in Scala).