CSC 347 - Concepts of Programming Languages

Static and Dynamic Types

Instructor: James Riely

Learning Objectives

❓ How should we support programmers in interpreting memory content correctly?

❓ How should we support programmers in applying operations correctly?

- Describe the role of types
- Identify and describe the effects of static and dynamic type checking

Programming Without Types

❓ What is difficult about the code below?

❓ How can we improve the language to better support programmers?

```
1 void* getNext(void* x) {
2   return *(void**)(x+7);
3 }
4 void* n = malloc(16);
5 *(int*)n = 42;
6 *(void**)(n+7) = NULL;
7 void* nn = getNext(n);
```

- Types!

```
 1 typedef struct Node {
 2   int value;
 3   Node* next;
 4 } Node;
 5
 6 Node* getNext (Node* x) {
 7   return x->next;
 8 }
 9
10 Node* n = malloc(sizeof(Node));
11 n->value = 42;
12 n->next = NULL;
13 Node* nn = getNext(n);
```

Programming Without Types

❓ How does Python overcome missing static types?

```
1 functools.reduce(function, iterable, [initial, ]/)
```

- Documentation: Python Library
- Examples
- Unit tests

Types

- Types define offsets

```
1 typedef struct Node {
2   int value;
3   Node* next;
4 } Node;
5
6 Node* getNext (Node* x) {
7   return x->next;
8 }
```

- Types determine valid operations

```
1 println( 1 - 2 )
2 println( "dog" - "cat" )
```

- Types are documentation

Type Enforcement

- Statically, track types with compiler
  - Compile time
  - Early
- Dynamically, store type with object
  - Run time
  - Late

Dynamic Type Checking

- Dynamic type checking resolves types or detects a failure at runtime
- Scheme is dynamic:
  - Accepted

```
1 #;> (define (f) (- 5 "hello"))
```

  - Fails at runtime

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

- Javascript is dynamic:
  - Accepted, but type resolves in surprising ways

```
1 var x = 5 - "hello"; // x === NaN
```

Static Type Checking in Java and Scala

Java

```java
int a = 5;
String b = "hello";
System.out.println ("Result = " + (a - b));
```

Scala

```scala
val a = 5
val b = "hello"
println(s"Result = ${a-b}")
```

- Compiler rejects code with `(5 - "hello")`

```
error: bad operand types for binary operator '-'
    System.out.println ("Result = " + (a - b));
                                        ^
  first type:  int
  second type: String
```

Dynamic Type Checking in Java and Scala

- We can make the error dynamic by casting

Java

```java
int a = 5;
String b = "hello";
System.out.println ("Result = " + (a - (int)(Object)b));
```

Scala

```scala
val a = 5
val b = "hello"
println(s"Result = ${a-b.asInstanceOf[Int]}")
```

- Compiler accepts code, but dynamic type checking at runtime catches the invalid cast

```
ClassCastException: class String cannot be cast to class Integer
```

- Can convert any static error into a dynamic error: casting turns compile-time errors into runtime errors

Variables in Static Languages

- In static languages, variables have types

Java

```java
int a = 5;
a = "hello";
```

Scala

```scala
var a = 5
a = "hello"
```

- Compiler error

```
incompatible types: String cannot be converted to int
    a = "hello";
        ^
```

- Type inference infers the most precise type possible

Variables in Dynamic Languages

- Variables do not have types in dynamic languages
- Only values have types

Scheme

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

Javascript

```
> function f() {
      var a = 5;
      a = "hello";
      console.log(a);
  }
> f()
"hello"
```

Type Conservativeness

Static type checking (Scala)

- Is sometimes conservative

```
1 def f(i: Int, s: String) = if true then i else s
```

- Runtime code modifications is harder
- Compiler optimizations based on types

Dynamic type checking (JavaScript)

- Not conservative: `f` always returns `i`

```
1 function f(i, s) = if true then i else s
```

- Runtime code modification often easy
- No compiler optimizations based on types

Summary

Static type checking

- At compile time
- Variables have types (often inferred)
- Benefits
  - compile-time detection of errors
  - no unit tests for type checking
  - automatic documentation
  - faster runtime (optimization)
  - less memory consumption

Dynamic type checking

- At runtime
- Variables do not have types
- Benefits
  - more flexible
  - usually conceptually simpler
  - faster compilation
  - easier runtime code generation/modification