

CSC 347 - Concepts of Programming Languages

Static and Dynamic Types

Instructor: Stefan Mitsch



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 *(x+4);  
3 }  
4 void* n = malloc(12);  
5 *n = 42;  
6 *(n+4) = 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

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

Scala

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

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

```
1 error: bad operand types for binary operator '-'  
2     System.out.println ("Result = " + (a - b));  
3                                     ^  
4     first type:  int  
5     second type: String
```



Dynamic Type Checking in Java and Scala

- We can make the error dynamic by casting

Java

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

Scala

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

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

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



Variables in Static Languages

- In static languages, variables have types

Java

```
1 int a = 5;  
2 a = "hello";
```

Scala

```
1 var a = 5  
2 a = "hello"
```

- Compiler error

```
1 incompatible types: String cannot be converted to int  
2     a = "hello";  
3     ^
```



Casting

- We can convert any static error into a dynamic one: casting turns compile-time errors into runtime errors

Java

```
1 int a = 5;  
2 a = (int)(Object)"hello";
```

Scala

```
1 var a = 5  
2 a = "hello".asInstanceOf[Int]
```

- Compiler accepts code, but invalid cast is detected at runtime

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



Type Inference

- Type inference infers the most precise type possible

Java

```
1 var a = 5;  
2 a = "hello";
```

Scala

```
1 var a = 5  
2 a = "hello"
```

```
1 error: incompatible types: String cannot be converted to int  
2     a = "hello";  
3         ^
```



Variables in Dynamic Languages

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

Scheme

```
1 #;> (define (main)
2       (define a 5)
3       (set! a "hello")
4       (display a)
5     )
6 #;> (main)
7 "hello"
```

Javascript

```
1 > function f() {
2     var a = 5;
3     a = "hello";
4     console.log(a);
5 }
6 > f()
7 "hello"
```



Static Type Checking

- Static types are conservative

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

- Static types may use type inference

```
1 val x = 1
```



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