

Types

CMSC 631

Reading

- Type Safety by Luca Cardelli
- Read Sections 1-3 by Tuesday
- Read Section 4-6 by Thursday
- If you really get into types
 - *Types and Programming Languages* by Benjamin Pierce

What is a type?

- A set of values
 - For example
 - integers that can be represented using 32-bit signed representation
 - floating point numbers represented using 64-bit IEEE-754 representation
 - Strings

more types

- 64-bit floats with units meters
- 64-bit floats with units feet
- 64-bit floats with units meters/second
- 32-bit integers representing a user id
- 32-bit integers representing length of arrays of wide characters
- 32-bit integers representing # of bytes

What types do

- Many functions & operators are only defined to work on certain types
 - e.g., integer addition
 - functions and operations might be overloaded
 - use either static or dynamic resolution
 - resolve '+' to integer addition, float addition or string concatenation

Type errors

- Applying a function to arguments of the wrong type is an error
- Resolved as:
 - static error
 - dynamic error
 - undefined behavior

What makes a programming language safe?

- An unsafe operation is one that is defined to have no defined meaning
 - Anything could happen
 - provide root access to script kiddies
 - Might be predictable on some platforms/implementations
 - for good or ill

Some unsafe behaviors

- Accessing outside the bounds of an array
- Using incorrect format string in scanf/printf
- using an integer as a pointer

Strong type systems

- Prevent undefined behavior due to use of incorrect types
- Typically coupled with measures to prevent other unsafe behaviors
- Generally involve substantial static component
 - some debate as to whether dynamic types really are a “type system”

What properties should a static type system have?

- Be verifiable
 - efficient type checking algorithm
 - JVM runs a type inference and checking algorithm every time it verifies a method
- Be transparent
 - programmer shouldn't be flummoxed by type checking algorithm
- Be enforceable

Why have static types?

- Efficient execution
 - don't need to check dynamic tag information
 - e.g., can use unboxed ints
- Detecting errors, small and large
 - using integer as a pointer
 - adding meters + feet
 - detecting errors statically is much better than detecting them during execution

why, continued

- Encourage better abstractions
- Better documentation
 - that is checked at each compilation

Type checking is theorem proving

- People throw a lot of effort and mathematical rigor at type checking
 - need to
- If you can confuse/mislead/subvert the type checker
 - you've just blown a security hole in your safe language

Before we dive into the math

- A review of some practical issues related to type checking in C++ and Java

Practical Subtyping

OO Programming

- OO Programming involves two very different concepts
 - inheritance - code reuse
 - in defining class B, I want to reuse method implementations defined for class A
 - subtyping - substitutivity
 - I want to be able to supply a B to someone who expects an A

Liskov substitution principle

- (Original?) Formal statement
 - If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .
- doesn't really handle interfaces or abstract classes
- Informal statement
 - If anyone expecting a T can be given an S , then S is a subtype of T

In an OO context, S subtype of T means

- The methods supported on instances of T are supported on instances of S
 - with compatible meanings
 - if the get method on T signals notFound by throwing an exception, S can't signal it by returning null

co and contra variant subtyping

- Forget method overloading on argument types for a moment
- class T {
 void foo(T t) { ... }
 T bar () { ... }
}
- class S subtype of T { ... }

Does this work?

- class T {
 void foo(T t) { ... }
 T bar () { ... }
}
- class S subtype of T {
 void foo(S t) { ... }
 S bar () { ... }
}

Partway

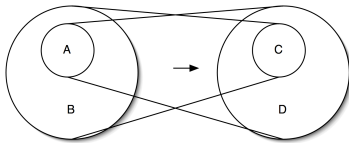
- foo doesn't
 - someone who expects a T could invoke foo and provide T as an argument
 - S's implementation of foo can't handle a T
- bar does
 - someone who expects a T could invoke bar and expect a T to be returned
 - should be able to handle an S

How about this?

- class T {
 void foo(S t) { ... }
 T bar () { ... }
}
- class S subtype of T {
 void foo(T t) { ... }
 S bar () { ... }
}

subtyping rules for functions

- When is $(B \sqsupseteq C) \sqsupseteq (A \sqsupseteq D)$?
- $A \sqsupseteq B$ (contravariant arguments)
- $C \sqsupseteq D$ (covariant return types)



Contravariants typing is rarely used/available

- Few, if any, programming languages use contravariant typing
 - argument types must match exactly in order to override
- C++ provides covariant return types
- Java does not
 - return types have to match exactly
 - a mistake, hard to fix now
- Java has covariant declared exceptions

Java errors

- class A implements Cloneable {
 public boolean equals(A a) { ... };
 public A clone() { ... };
 ...
}

Pre and Post conditions

- class T {
 // Precondition P
 // Postcondition Q
 ... foo(...) {...}
}
- class S is a subtype of T {
 // Precondition P'
 // Postcondition Q'
 ... foo(...) {...}
}
- P must imply P'
 – S can relax preconditions of T
- Q' must imply Q
 – S can guarantee more than T
- $P \supseteq P'$ and $Q' \supseteq Q$

types and conditions similar

- When can Q:R foo(P:A)
- be overridden by Q':R' foo'(P':A')
- $P \supseteq P'$ and $Q' \supseteq Q$
 – $P \supseteq P'$ and $Q' \supseteq Q$
- $A \supseteq A'$ and $R' \supseteq R$

Checked Exceptions

- In Java, declared exceptions are checked at compile time
 - all checked exceptions must be declared
 - runtime exceptions and errors don't need to be
- In C++, if a method declares that it throws exceptions
 - runtime check that no other errors are thrown

subtyping of declared exceptions

- which gives an error?
- class T {
 void foo() throws IOException { ... };
 void bar() { ... };
}
- class S extends T {
 void foo() { ... };
 void bar() throws IOException { ... };
}

- Given

```
/** Search array for value */  
/** @precondition: a is sorted */  
/** @postcondition: returns index i s.t. a[i] == value,  
    or -1 if no such value exists */  
int search( int [] a, int value);
```
- In an overriding function can we
 - a) Change the precondition to true?
 - b) Change the precondition to a is sorted and there exists an i s.t. a[i] = value?
 - c) Change the postcondition so that i == -1 or i is the first index s.t. a[i] == value?
 - d) Change the function so that it throws "NoSuchElementException" rather than returning -1 when value does not occur in a.

Liskov substitution principle

- All comes down to Liskov substitution principle
 - If you can intuitively apply Liskov substitution principle,
 - you understand how it should work
 - it doesn't always work the way it *should* work

C++ weirdness

- ```
class A {
 public: void foo(int x);
}
```
- ```
class B : public A {
    public: void foo(char * s);
}
```
- ```
B b; A * ap = &b;
```
- ```
ap->foo(42);
```
- ```
b.foo(42);
```

no matching function for call to `B::foo(int)`  
candidates are: void B::foo(char \*)

## C++ speculation

- Weirdness designed to provide compile-time covariant argument types
- ```
class A {
    public: A & operator=(const A & a) {...};
}
```
- ```
class B : public A {
 public: B & operator=(const B & b) {...};
}
```
- ```
B b; A a; A* ap = &b;
```
- ```
b = a; // generate compile time error
```
- ```
*ap = a; // doesn't generate error
```

Subtyping of arrays

- if $A \sqsubseteq B$, is $(\text{array of } A) \sqsubseteq (\text{array of } B)$?
- Allows poor-man's polymorphism
 - public void reverse(Object [] a)
- Dubious idea

Java errors

- ```
void foo(Object[] a) {
 a[1] = new Integer(42);
};
```
- ```
String [] test = ["x", "y", "z"];
foo(test);
String x = test[1];
```

Run time store checks

- Java arrays of references use run time store checks
 - If storing a object of type T into an array instance A
 - check at run-time that the type specified when creating A is a supertype of T
- Performance overhead
- Runtime errors are bad
 - but could be worse (*see* C++)

C++ array subtyping

```
struct A {  
    int x;  
};  
struct B : public A {  
    int y;  
};  
void foo(A a[]) {  
    a[1].x = 42;  
};  
B b[5];  
foo(b);
```

C++ store subtyping

- `class A { ... };`
- `void swap(A & a1, A & a2) {
 A tmp = a1;
 a1 = a2;
 a2 = tmp;
}`
- `B b; C c; class B : public A { ... };`
- `class C : public A { ... };`
- `swap (b,c);`