



UCF

**College of Engineering  
and Computer Science**

UNIVERSITY OF CENTRAL FLORIDA

# Types

COP-3402 Systems Software  
Paul Gazzillo



UCF

# Why Use Types?

To prevent errors during runtime

# Typed vs Untyped

A type is

- a set of values
- and operations on those values
- int: set of integers and the arithmetic operations
- bool: true/false and the logic connectives (and, or, not)

Typed languages restrict variable's range of values (Python, C, Java, etc)

Untyped languages do not (Lisp, assembly)

# Safe vs Unsafe

Runtime errors are

- Trapped
  - terminated by machine, e.g., NULL-pointer error, divide-by-zero
- Untrapped
  - program continues, e.g., write past array bounds

Safe languages prevent untrapped (and some trapped) errors

# Static vs Dynamic Checking

When do checks happen

- Compile-time (static): C, Java
- Run-time (dynamically): Python, Java(?)

# Weak vs Strong

Forbidden errors: all untrapped errors and some trapped errors

Good behavior: a program has no forbidden behaviors

- Strongly-checked: all legal programs have good behavior
- Weakly-checked: some programs violate safety

**Table 1. Safety**

	Typed	Untyped
Safe	ML, Java	LISP
Unsafe	C	Assembler

<http://lucacardelli.name/Papers/TypeSystems.pdf>

# Demo: Python vs C



# Static Type Checking

- Record (or infer) types of identifiers in symbol table
- Post-order tree traversal
- Check identifiers used in
  - Arithmetic operators
  - Function calls
  - Assignments
- Lookup type in symbol table
- Constants have a fixed type
  - 3 is an int
  - 5.2 is a float
  - True is a bool (note: C does not have a bool type)

# Function Types

- Scalar values have a *primitive* type
  - int, char, long, etc
- If symbol "x" has type "int" we can write:

$x : \text{int}$

- Function types describe parameters and return values
- If f takes two integers and return a bool, we can write

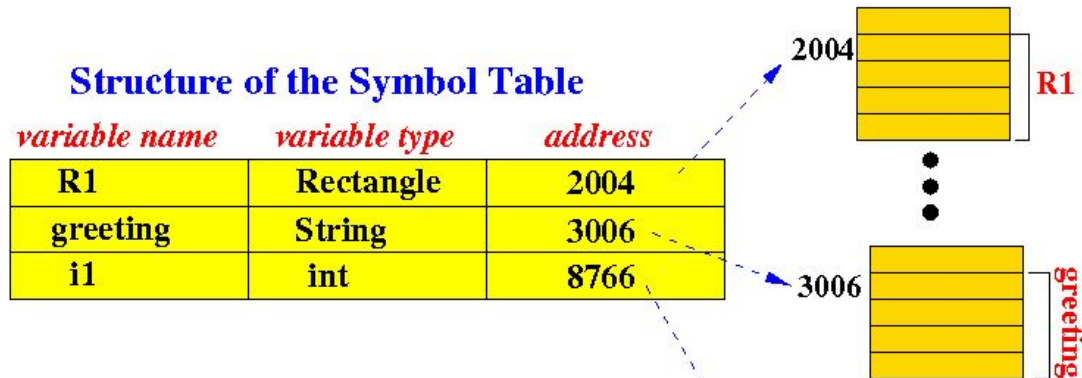
$f : (\text{int}, \text{int}) \rightarrow \text{bool}$

- What is the type of multiplication (\*)?

$* : (\text{int}, \text{int}) \rightarrow \text{int}$

# Symbol Table: Mapping Variables to Memory

- Compiler assigns memory to each variable
- Maintains mapping between names and locations
- Creates new mapping on declaration
- Refers to mapping when variables are used



# Demo: Static Checking a Tree

```
int x;  
int y;  
read x;  
y = 1 + x * 7;
```

```
int x;  
bool y;  
read x;  
y = 1 + x;  
print y * (x + 1)
```

# Safety Guarantees

If a type checker accepts a program is it actually safe?

*type soundness*: checker says safe, program is safe

Example: memory corruption due to index out of bounds

- unsound: C type checker permits the program
- sound: Java type checker rejects the program (at runtime)

# Proving Type Soundness

Goal: well-typed programs are safe programs

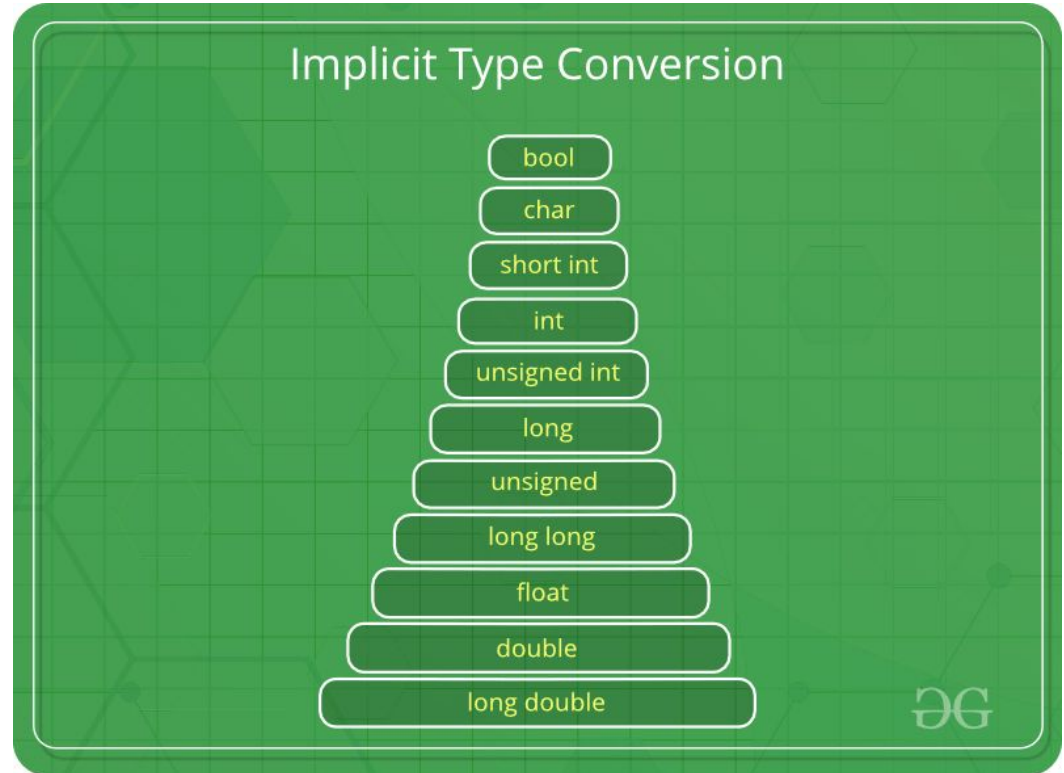
Formal soundness: each provable sentence is valid with respect to semantics

Need to define semantics first

Define type rules that "run" over the semantics

# Type Coercion in C

- Instead of type error
- C inserts conversions
- Converts to highest-precision type
- `char + int -> int + int`



# Demo: Subverting C's Type System