College of Engineering and Computer Science
UNIVERSITY OF CENTRAL FLORIDA

Types

COP-3402 Systems Software
Paul Gazzillo

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

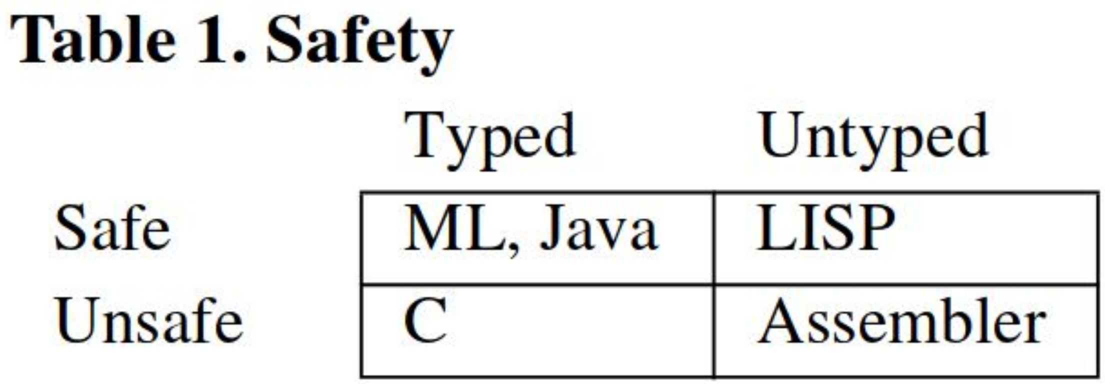


Table 1. Safety

	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

Demo: Static Checking a Tree

```
int x;
int y;
read x;
read y;
print 1 + x * (7 + y);
```

```
int x;
bool y;
read x;
read y;
print 1 + x * (7 + y);
```

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: <u>well-typed programs</u> are <u>safe programs</u>

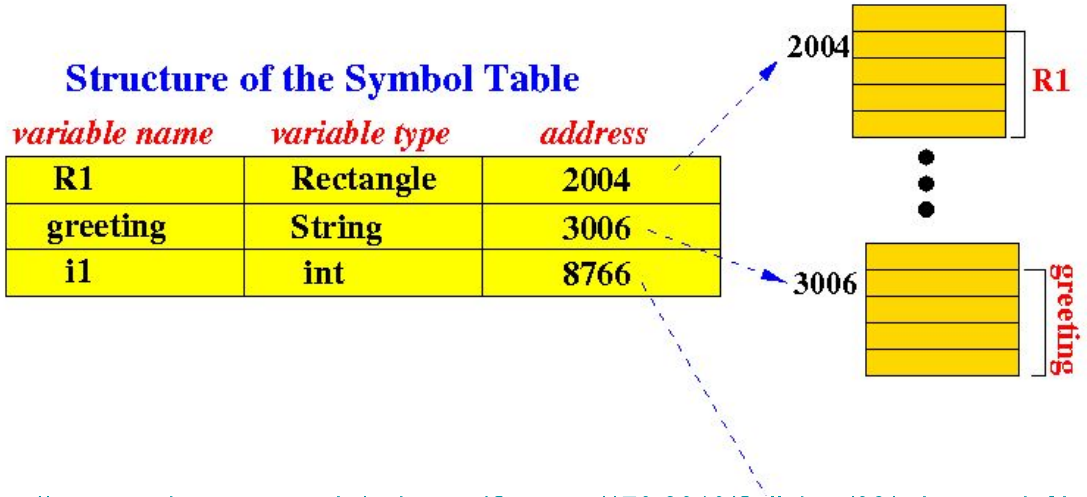
Formal soundness: each <u>provable sentence</u> is <u>valid with respect to semantics</u>

Need to define semantics first

Define type rules that "run" over the semantics

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



Structure of the Symbol Table

variable name	*variable type*	*address*
R1	Rectangle	2004
greeting	String	3006
i1	int	8766

http://www.mathcs.emory.edu/~cheung/Courses/170.2010/Syllabus/02/prim-var-def.html

Demo: Symbol Tables

SimpleC Project 2 Only Has One Type

- No need to implement true type checking
  - Just check for undefined symbols
- Symbol table only needs name and LLVM IR var
  - Later symbol table will be extended for functions
- Symbol tables are dictionaries: map from key to value
  - Linked list
  - Hash table
  - Dynamic array