

Intro: Type Theory

CS 6V81-002: Language-based
Security

January 14, 2008

Types as Abstractions

- Types – abstractions of program values
- Examples:
 - type “int” abstracts the set $-2^{31}+1 .. 2^{31}$
 - type “bool” abstracts the set {true,false}
 - Note: “false” and “0” are *different values*
- Code can be a value
 - type “int \rightarrow bool” abstracts set of functions mapping an int argument to a bool return value
 - Example: $f:\text{int}\rightarrow\text{bool}$, $x:\text{int}$, $f(x):\text{bool}$
- Some values have more than one type
 - Example: class hierarchies
 - parent classes denote higher abstractions (less info)

Type Systems

- **Typing Rules** define types for expressions

– Example:

$$\frac{e_1 : \tau_1 \rightarrow \tau_2 \quad e_2 : \tau_1}{e_1(e_2) : \tau_2}$$

- An expression is **well-typed** if there exists a typing derivation that assigns it a type
- **type-inference** – a procedure for finding a type (and typing derivation) for an arbitrary well-typed expression
- **type-checking** – given an expression and a type, verify that the expression has that type
 - Even easier if you give the type-checker a proposed typing derivation!

Uses for Type Systems

- Help programmer find bugs
 - Examples: `3>true`, `x:=12` when `x` has type `bool`
- Help compiler discover optimizations
 - Example: `((LeafClass) foo).bar()`;
- Prevent illegal runtime operations
 - Example: `foo.bar()`; when `foo` has no method `bar`
- Language-based Security
 - define security violations as “illegal runtime operations”
 - well-typed programs are secure
 - typing derivation is a proof of security adherence!
 - harder when source code not disclosed: need to extend source-level typing info down to object code

Properties of Type Systems

- **Soundness** – well-typed programs should never perform illegal operations at runtime
 - **operational semantics**: define all legal operations (“steps”)
 - **subject reduction**: each step preserves the program’s type
 - **progress**: each well-typed program are either done or can take a step
- **Completeness** – for any computable function, there should exist a well-typed program that computes it
- Tractability issues
 - type-checking should be computable (usually linear)
 - type-inference might not be (use programmer annotations)