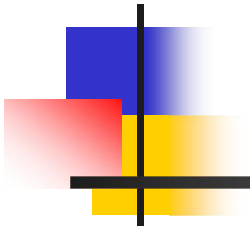


Programming Languages and Compilers (CS 421)



Reza Zamani

<http://www.cs.illinois.edu/class/cs421/>

Based in part on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha and Elsa Gunter



Why Data Types?

- n Data types play a key role in:
 - n *Data abstraction* in the design of programs
 - n *Type checking* in the analysis of programs
 - n *Compile-time code generation* in the translation and execution of programs



Terminology

- n Type: A type t defines a set of possible data values
 - n E.g. **short** in C is $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
 - n A value in this set is said to have type t
- n Type system: rules of a language assigning types to expressions



Types as Specifications

- n Types describe properties
- n Different type systems describe different properties, eg
 - n Data is read-write versus read-only
 - n Operation has authority to access data
 - n Data came from “right” source
 - n Operation might or could not raise an exception
- n Common type systems focus on types describing same data layout and access methods



Sound Type System

- n If an expression is assigned type t , and it evaluates to a value v , then v is in the set of values defined by t
- n SML, OCAML, Scheme and Ada have sound type systems
- n Most implementations of C and C++ do not



Strongly Typed Language

- n When no application of an operator to arguments can lead to a run-time type error, language is *strongly typed*
 - n Eg: `1 + 2.3;;`
- n Depends on definition of “type error”



Strongly Typed Language

- n C++ claimed to be “strongly typed”, but
 - n Union types allow creating a value at one type and using it at another
 - n Type coercions may cause unexpected (undesirable) effects
 - n No array bounds check (in fact, no runtime checks at all)
- n SML, OCAML “strongly typed” but still must do dynamic array bounds checks, runtime type case analysis, and other checks



Static vs Dynamic Types

- *Static type*: type assigned to an expression at compile time
- *Dynamic type*: type assigned to a storage location at run time
- *Statically typed language*: static type assigned to every expression at compile time
- *Dynamically typed language*: type of an expression determined at run time



Type Checking

- n When is $\text{op}(\text{arg1}, \dots, \text{argn})$ allowed?
- n *Type checking* assures that operations are applied to the right number of arguments of the right types
 - n Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- n Used to resolve overloaded operations



Type Checking

- n Type checking may be done *statically* at compile time or *dynamically* at run time
- n Dynamically typed (aka untyped) languages (eg LISP, Prolog) do only dynamic type checking
- n Statically typed languages can do most type checking statically



Dynamic Type Checking

- n Performed at run-time before each operation is applied
- n Types of variables and operations left unspecified until run-time
 - n Same variable may be used at different types



Dynamic Type Checking

- n Data object must contain type information
- n Errors aren't detected until violating application is executed (maybe years after the code was written)



Static Type Checking

- n Performed after parsing, before code generation
- n Type of every variable and signature of every operator must be known at compile time



Static Type Checking

- n Can eliminate need to store type information in data object if no dynamic type checking is needed
- n Catches many programming errors at earliest point
- n Can't check types that depend on dynamically computed values
 - n Eg: array bounds



Static Type Checking

- n Typically places restrictions on languages
 - n Garbage collection
 - n References instead of pointers
 - n All variables initialized when created
 - n Variable only used at one type
 - n Union types allow for work-arounds, but effectively introduce dynamic type checks