

# CS421 Topic 6: Overview of Type Systems<sup>1</sup>

Sameer Sundresh  
sundresh@uiuc.edu

University of Illinois at Urbana-Champaign

June 4, 2007

---

<sup>1</sup>Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, Elsa Gunter, and Mark Hills

# Introduction: Why Have Data Types?

Data types play a key role in:

- ▶ the design of programs (through data abstraction);
- ▶ the analysis of programs (through type checking and type inference);
- ▶ the translation and execution of programs (through compile-time code generation).

Underlying these uses are type systems that provide one way to express the meaning of a program.

# Objectives

## Terminology of types and type checking

- ▶ soundness
- ▶ strong and weak typing
- ▶ static and dynamic type checking

# Type

A type  $t$  defines a set of possible data values

- ▶ E.g., short in C  $\equiv \{i : -2^{15} \leq i \leq 2^{15} - 1\}$
- ▶ A value in this set is said to be of type  $t$ .

# Type System

The rules of a language that describe how to assign a type to each expression in a program are the *type system*.

# Sound Type Systems

A type system is **sound** if the type  $t(e)$  it assigns to expression  $e$  is always correct, i.e., whenever  $e$  is assigned type  $t$  and evaluates to value  $v$ , the value  $v$  is in the set of values defined for type  $t$

- ▶ SML, OCAML, and Ada have sound type systems.
- ▶ Most implementations of C and C++ do not.

## Type Errors

An operation that is applied to an operand of an illegal type is a *type error*. When the operation may be applied to an operand of an illegal type, this is a *potential type error*.

- ▶ `1 +. 2.5` *in OCAML*
- ▶ `atoi();` *in C*
- ▶ `MyCls cref = (MyCls) X;` *in Java*

This last may or may not be a type error. Such an error must be detected at run time.

## Strongly Typed Languages

A language is **strongly typed** when no application of an operator to its arguments can lead to a run-time type error. This depends strongly on the definitions of “type” and “type system” for the language. A sound type system is required, and expensive checks may be needed.

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

# Weakly Typed Languages

A language is **weakly typed** when applications of operators to operands *can* lead to run-time type errors or, worse, undetectable errors that do not cause the program to halt.

- ▶ E.g., Perl, C, C++, Fortran90, assembly languages
- ▶ Advantages: more flexible programming; faster code

# Statically Typed Languages

A **static type** is a type assigned to an expression at compile-time.  
A **statically typed language** is one where every expression is assigned a static type.

- ▶ **Weakly Statically Typed:** Assigned types are *assumed to be correct* and not checked.  
E.g., C, C++, Fortran90
- ▶ **Strongly Statically Typed:** Assigned types are checked *at compile-time or run-time*  
E.g., ML, Haskell, Ada, OCAML, Java.

## Dynamically Typed Language

A **dynamic type** is a type assigned to a storage location at run time. A **dynamically typed language** is one where the type of an expression may not be determined until run time.

- ▶ Safe, but may incur significant run-time overheads
- ▶ Cost: Cannot generate unique code for some operations at compile-time
  - ⇒ Generally used only in interpreted languages

*Examples:* Lisp, Scheme, SmallTalk, Python

# Type Checking

*Type checking* is a program analysis that attempts to check whether a program can generate type errors.

- ▶ Type checking assures that operations are applied to the right number of arguments of the right types
- ▶ Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- ▶ Used to resolve overloaded operations

# Type Checking

Type checking may be done *statically* at compile time or *dynamically* at run time.

- ▶ dynamically typed (aka untyped) languages, like Lisp and Prolog, do only dynamic type checking
- ▶ statically typed languages can do most type checking statically (at compile time)

# Dynamic Type Checking

- ▶ dynamic type checking is performed at run-time before each operation is applied
- ▶ types of variables and operations are left unspecified until run-time
- ▶ variables may be used at different times with different types
- ▶ data objects must maintain type information at runtime
- ▶ type errors are not detected until the violating section of code is executed, sometimes years after being written

## Dynamic Types: An Example (Python)

```
1 >>> x = 7
2 >>> def f(): return x + y
3 ...
4 >>> y = "three"
5 >>> f()
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in ?
8   File "<stdin>", line 1, in f
9 TypeError: unsupported operand types for +: 'int' and 'str'
10 >>>
```

## Static Types: An Example (OCaml)

```
1 # let x = 7;;  
2 val x : int = 7  
3 # let f _ = x + y;;  
4 Unbound value y  
5 # let y = "three";;  
6 val y : string = "three"  
7 # let f _ = x + y;;  
8 This expression has type string but is here used with type int  
9 # f ();;  
10 Unbound value f  
11 #
```

# Static Type Checking

- ▶ performed after parsing, but before code generation – considered part of the compiler *front-end*
- ▶ type of every variable and type signature of every operator must be known at compile time
- ▶ can eliminate need to store type information with data object if no dynamic checks are needed – *type erasure*
- ▶ catches many programming errors early – before execution
- ▶ cannot check types that depend on dynamically computed values, like array bounds

# Static Type Checking

Static type checking typically places some restrictions on the language:

- ▶ garbage collection instead of explicit allocation and deallocation
- ▶ references instead of pointers
- ▶ variables initialized automatically when created
- ▶ variables only used at one type (unions allow workarounds, but introduce dynamic checks)

## Type Checking Costs

Type checking may incur several possible run-time overheads:

1. Run-time type checks, e.g., `MyCls cref = (MyCls) X;`
2. Run-time array bounds checks, e.g., `int X = A[i];`
3. Run-time null pointer checks, e.g., `int X = *p;`
4. Proper initialization of data values, e.g., zeroing in `new T;`
5. Garbage Collection (GC): disallow explicit `free / delete`

**Static type checking** may eliminate some instances of the first 3 kinds of checks. #2 and #3 are especially hard to eliminate and every widely-used, strongly typed language relies on GC!

# Type Declarations

A *type declaration* is the explicit assignment of a type to a variable or function in the program source

- ▶ must be checked in a strongly typed language
- ▶ often not necessary for strong typing or even static typing

# Type Inference

A program analysis that attempts to identify the type of an expression from the program context of the expression; also called *type reconstruction*

- ▶ fully static type inference first introduced by Robin Milner in ML
- ▶ Haskell, OCaml, and SML all use type inference
- ▶ As we've noticed, records are a problem for type inference

# Type Equivalence and Compatibility

Two types that represent the same underlying data items are said to be *equivalent*. Equivalence comes in two forms:

- ▶ Name equivalence, where two types are equivalent if they share the same name
- ▶ Structural equivalence, where two types are equivalent if they have the same structure

Sometimes equivalence is not necessary. In these cases, operations may just need a *compatible* type.

## Type Casts and Coercions

Data can often be converted from one type to another.

- ▶ *Casting* occurs when data of one type is explicitly converted to another type
- ▶ *Nonconverting type casts* reinterpret the underlying bits as a different data type, but do not perform any conversion on the data
- ▶ *Coercions* occur behind the scenes when data of one type is changed to another to make it compatible with an operation on the data (i.e. converting int to double in C floating-point addition)

# Advantages of Strong Typing

## Engineering

- ▶ A program that type-checks is likely to have fewer errors
- ▶ Types constrain the use of a function

## Safety

- ▶ Ever try casting an integer into a pointer in C?

```
1 int i = someIntFunc();  
2 *((int *)i) = 10;
```

- ▶ A type error in a weakly typed language can be very hard to debug.

## Theory

- ▶ It is much easier to verify the operation of type-correct code.

# Types as Specifications

Type systems are a major area of research, and can be very involved, moreso than we have shown here. Types can be seen as *specifications* of desired program behavior.

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