

Objectives

CS421 Topic 6: Overview of Type Systems¹

Sameer Sundresh
sundresh@uiuc.edu

University of Illinois at Urbana-Champaign

June 4, 2007

¹Based 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.

Type

Type System

Sound Type Systems

- 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 .

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

- 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.

<p>Introduction and Objectives Type System Basics Classes of Type Systems By Checking Types Deductions Applications</p>	<p>Type System Type Errors</p>	<p>Strongly Typed Languages Weakly Typed Languages Type Checking Dynamically Typed Languages</p>	<p>Introduction and Objectives Type System Basics Classes of Type Systems By Checking Types Deductions Applications</p>
<h2>Type Errors</h2> <p>An operation that is applied to an operand of an illegal type is a <i>type error</i>. When the operation <i>may</i> be applied to an operand of an illegal type, this is a <i>potential type error</i>.</p> <ul style="list-style-type: none"> ▶ 1 +, 2.5 <i>in OCAML</i> ▶ atoi(0); <i>in C</i> ▶ MyClass cref = (MyCls) X; <i>in Java</i> <p>This last may or may not be a type error. Such an error must be detected at run time.</p>	<h2>Strongly Typed Languages</h2> <p>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.</p> <ul style="list-style-type: none"> ▶ C++ claimed to be “strongly typed”, but <ul style="list-style-type: none"> ▶ 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 	<h2>Weakly Typed Languages</h2> <p>A language is weakly typed when applications of operators to operands <i>can</i> lead to run-time type errors or, worse, undetectable errors that do not cause the program to halt.</p> <ul style="list-style-type: none"> ▶ E.g., Perl, C, C++, Fortran90, assembly languages ▶ Advantages: more flexible programming: faster code 	<p>Sameer Suresh CS421, Topic 6: Overview of Type Systems</p>
<p>Introduction and Objectives Type System Basics Classes of Type Systems By Checking Types Deductions Applications</p>	<p>Strongly Typed Languages Weakly Typed Languages Type Checking Dynamically Typed Languages</p>	<p>Introduction and Objectives Type System Basics Classes of Type Systems By Checking Types Deductions Applications</p>	<p>Sameer Suresh CS421, Topic 6: Overview of Type Systems</p>
<h2>Statically Typed Languages</h2> <p>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.</p> <ul style="list-style-type: none"> ▶ Weakly Statically Typed: Assigned types are <i>assumed to be correct</i> and not checked. E.g., C, C++, Fortran90 ▶ Strongly Statically Typed: Assigned types are checked at <i>compile-time or run-time</i> E.g., ML, Haskell, Ada, OCAML, Java. 	<h2>Dynamically Typed Language</h2> <p>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.</p> <ul style="list-style-type: none"> ▶ 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 <p><i>Examples:</i> Lisp, Scheme, SmallTalk, Python</p>	<h2>Type Checking</h2> <p><i>Type checking</i> is a program analysis that attempts to check whether a program can generate type errors.</p> <ul style="list-style-type: none"> ▶ 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 	<p>Sameer Suresh CS421, Topic 6: Overview of Type Systems</p>

Introduction and Objectives
Type System Basics
Classes of Type Systems
Type Checking
Type Deductions / Applications

Type Checking
Dynamic Type Checking
Static Type Checking

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)

Introduction and Objectives
Type System Basics
Classes of Type Systems
Type Checking
Type Deductions / Applications

Type Checking
Dynamic Type Checking
Static Type Checking

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

Introduction and Objectives
Type System Basics
Classes of Type Systems
Type Checking
Type Deductions / Applications

Type Checking
Dynamic Type Checking
Static Type Checking

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 >>>

```

Introduction and Objectives
Type System Basics
Classes of Type Systems
Type Checking
Type Deductions / Applications

Type Checking
Dynamic Type Checking
Static Type Checking

Static Type Checking

```

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 #

```

- ▶ 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

Introduction and Objectives
Type System Basics
Classes of Type Systems
Type Checking
Type Deductions / Applications

Type Checking
Dynamic Type Checking
Static Type Checking

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)

Introduction and Objectives
 Type System Basics
 Classes of Type Systems
Type Checking
 Type Declarations & Coercions
 Applications

Type Checking
 Dynamic Type Checking
 Static Type Checking

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!

Samer Sumrish CS421 Topic 6: Overview of Type Systems

Introduction and Objectives
 Type System Basics
 Classes of Type Systems
Type Declarations & Coercions
 Applications

Type Declarations
 Type Inference
 Type Casting and Compatibility
 Type Casts and Coercions

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

Samer Sumrish CS421 Topic 6: Overview of Type Systems

Introduction and Objectives
 Type System Basics
 Classes of Type Systems
Type Declarations & Coercions
 Applications

Type Inference
 Type Casting and Compatibility
 Type Casts and Coercions

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

Samer Sumrish CS421 Topic 6: Overview of Type Systems

Introduction and Objectives
 Type System Basics
 Classes of Type Systems
Type Declarations & Coercions
 Applications

Type Declarations
 Type Inference
Type Equivalence and Compatibility
 Type Casts and Coercions

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.

Introduction and Objectives
 Type System Basics
 Classes of Type Systems
Type Declarations & Coercions
 Applications

Type Declarations
 Type Inference
 Type Casting and Compatibility
 Type Casts and Coercions

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)

Introduction and Objectives
 Type System Basics
 Classes of Type Systems
Type Declarations & Coercions
 Applications

Type Declarations
 Type Inference
 Type Casting and Compatibility
 Type Casts and Coercions

Advantages of Strong Typing

Engineering

- ▶ A program that type-checks is likely to have fewer errors
- ▶ Types constrain the use of a function
- ▶ Ever try casting an integer into a pointer in C?

Safety

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

Theory

- ▶ A type error in a weakly typed language can be very hard to debug.
- ▶ It is much easier to verify the operation of type-correct code.

Samer Sumrish CS421 Topic 6: Overview of Type Systems

Samer Sumrish CS421 Topic 6: Overview of Type Systems

Samer Sumrish CS421 Topic 6: Overview of Type Systems

Types as Specifications

Type systems are a major area of research, and can be very involved, more so 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