Advanced Programming Methods

Lecture 8 - Type Systems, Type Checker

What is a Type?

Set of values and operations allowed on those values

- Integer is any whole number in the range $-2^{31} \leq i < 2^{31}$
- `enum colors = {red, orange, yellow, green, blue, black, white}`

Few types are predefined by the programming language

Other types can be defined by a programmer

- Declaration of a structure in C

What is a Type?

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type `integer`, then the result is of type `integer`

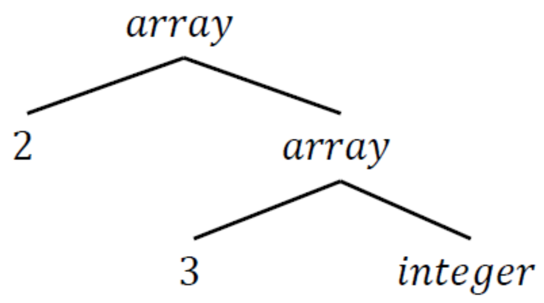
Each expression has a type associated with it

Type Expression

- Type of a language construct is denoted by a type expression

- A basic type is a type expression
- A type constructor operator applied to a type expression is a type expression

```
int a[2][3];
```



Type System

- The set of types and their associated rules are collectively called a type system
    - Rules to associate type expressions to different parts of a program (for e.g., variables, expressions, and functions)
- Type systems include rules for type equivalence, type compatibility, and type inference

Type Checking

- Ensure that valid operations are invoked on variables and expressions
    - `&&` operator in Java expects two operands of type boolean
- Means both type inferencing and identifying type-related errors
    - A violation of type rules is called type clash
    - Errors like arithmetic overflow is outside the scope of type systems
        - Run-time error, not a type clash
    - Can catch errors, so needs to have a notion for error recovery

- A type checker implements a type system

Type Error in Java

```
class A {
  int add1(int x) {
    return x + 1;
  }

  public static void main(String
args[]) {
    A a = new A();
    if (false)
      add1(a);
  }
}
```

```
TypeError.java:9: error: incompatible
types: A cannot be converted to int
              add1(a);
                   ^
Note: Some messages have been simplified;
recompile with -Xdiags:verbose to get
full output
1 error
```

Base Types

- Modern languages include types for operating on numbers, characters, and Booleans
  - Similar to operations supported by the hardware
- Individual languages may add additional types
  - Exact definitions and types vary across languages
  - C does not have the string type
- There are two additional basic types
  - void: no type value
  - type error: error during type checking

Constructed Types

- Constructed types are created by applying a type constructor to one or more base types
  - Also called nonscalar types
  - Examples are arrays, strings, enums, structures, unions

Constructed Types

Structure	Union

`struct Node {`
`  struct Node* next;`
`  int value;`
`};`
	`union Data {`
	`  int i;`
	`  float f;`
	`  char str[16];`
	`};`

Type of `Node` may be `(Node*) x int`	Type of `Data` may be `int` U `float` U `char[]`

Type Constructors

- If T is a type expression then $array(I, T)$ is a type expression denoting the type of an array with elements of type T and index set I
  - I is often a range of integers

    ```
    int A[10];
    ```
  - A can have type expression $array(0\,..\,9, integer)$
    - C uses equivalent of `int*` as the array type

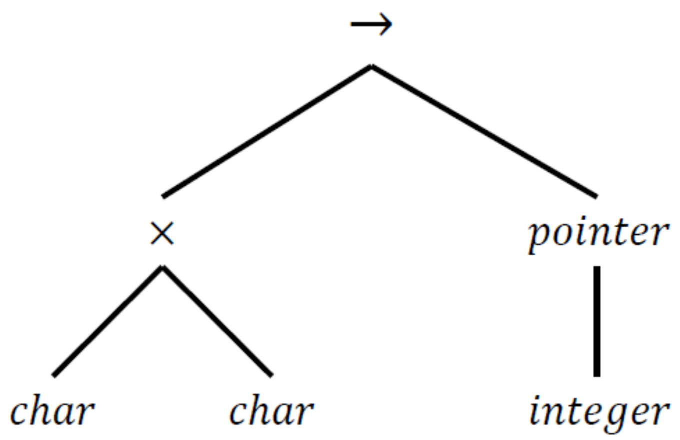
- If T_1 and T_2 are type expressions, then the Cartesian product $T_1 \times T_2$ is a type expression

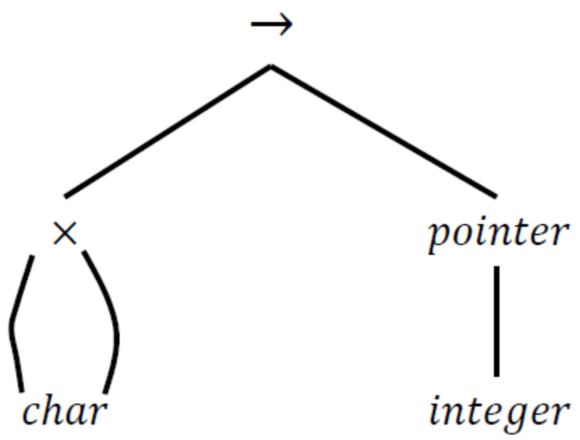
Type Constructors

- Function maps domain set to a range set
  - Denoted by type expression $D \rightarrow R$
  - Type of function `int* f(char a, char b)` is denoted by `char` $\times$ `char` $\rightarrow$ `int*`

Type Constructors

$$char \times char \rightarrow pointer(integer)$$





Pointer Types

- If T is a type expression then $pointer(T)$ is a type expression denoting type pointer to an object of type T

Other Classifications

- Scalar and compound types
  - Discrete, rational, real, and complex types constitute the scalar types
  - Scalar indicates a single value, also called simple types
  - Example of compound types are arrays, maps, sets, and structs
  - String in C is a compound type
- Primitive and reference types
  - Is this directly a value, or is it a reference to something that contains the real value?

Polymorphism

Polymorphism

- Use a single interface for entities of multiple types
  - Applicable to both data and functions
  - A function that can operate on arguments of different types is a polymorphic function
  - Built-in operators for indexing arrays, applying functions, and manipulating pointers are usually polymorphic

- Ad hoc, parametric and subtype polymorphism
- Static and dynamic polymorphism

Ad hoc Polymorphism

- Explicitly specifies the set of types
- Refers to polymorphic functions that can be applied to arguments of different types
    - Behavior depends on the type of the argument
- E.g., function and operator overloading

```
String fruits = "Apple" + "Orange";

int a = b + c;
```

Parametric Polymorphism

- Code takes type or set of types as parameter, either explicitly or implicitly
- Parametric polymorphism does not specify the exact types
  - Type of the result is a function of the argument types
- Explicit parametric polymorphism is called generics or templates
  - Used mostly in statically-typed languages

```
class List<T> {
  class Node<T> {
    T elem;
    Node<T> next;
  }
  Node<T> head;
  ...
}

List<B> map(Func<A, B> f, List<A> x) {
    ...
}
```

Subtype Polymorphism

- Used in object-oriented languages
  - Code is designed to work with values of some specific type T
  - Programmer can define extensions of T to work with the code

```
abstract class Animal {
    abstract String talk();
}
class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}
class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}
void main(String[] args) {
    (new Cat()).talk();
    (new Dog()).talk();
}
```

Static and Dynamic Polymorphism

- Static polymorphism – which method to invoke is determined at compile time
  - For e.g., by checking the method signatures (method overloading)
  - Usually used with ad hoc and parametric polymorphism
- Dynamic polymorphism – wait until run time to determine the type of the object pointed to by the reference to decide the appropriate method invocation
  - For e.g., by method overriding
  - Usually used with subtype polymorphism

Type Equivalence

Are these definitions the same?

```
struct Tree {
    struct Tree *left;
    struct Tree *right;
    int value;
};
```

```
struct STree {
    struct STree *left;
    struct STree *right;
    int value;
};
```

Are these definitions the same?

- Should the reversal of the order of the fields change type?
  - Some languages say no, most languages say yes

```
type R1 = record
            a, b : integer
end;
```

```
type R2 = record
            a : integer
            b : integer
end;
```

```
type R3 = record
            b : integer
            c : integer
end;
```

```
type str = array [1...10] of char;
```

```
type str = array [0...9] of char;
```

Type Equivalence

- Mechanism to decide the equivalence of two types

- Two approaches
  - Nominal equivalence – two type expressions are same if they have the same name (e.g., C++, Java, and Swift)
  - Structural equivalence – two type expressions are equivalent if
    I. Either both are the same basic types,
    II. or, are formed by applying same type constructor to equivalent types

Type Equivalence

Nominal	Structural

`class Foo {`
`  method(input: string): number { ... }`
`}`
`class Bar {`
`  method(input: string): number { ... }`
`}`
`let foo: Foo = new Bar(); // ERROR`
	`class Foo {`
	`  method(input: string): number { ... }`
	`}`
	`class Bar {`
	`  method(input: string): number { ... }`
	`}`
	`let foo: Foo = new Bar(); // Okay`

Type Equivalence

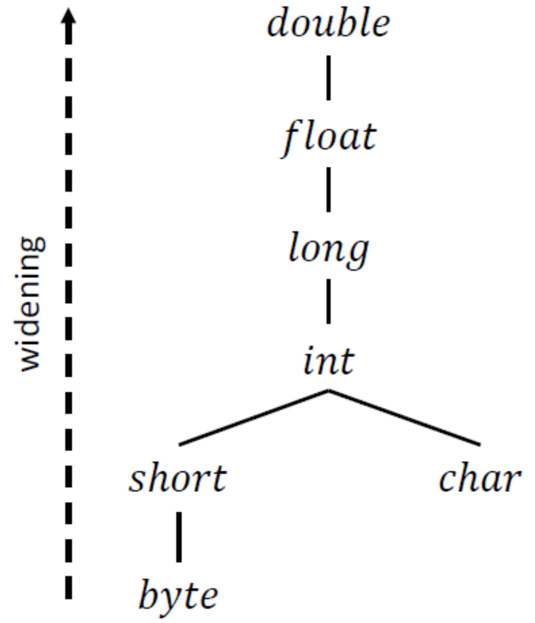
Nominal	Structural
• Equivalent if same name • Identical names can be intentional • Can avoid unintentional clashes • Difficult to scale for large projects	• Equivalent only if same structure • Assumes interchangeable objects can be used in place of one other • Problematic if values have special meanings

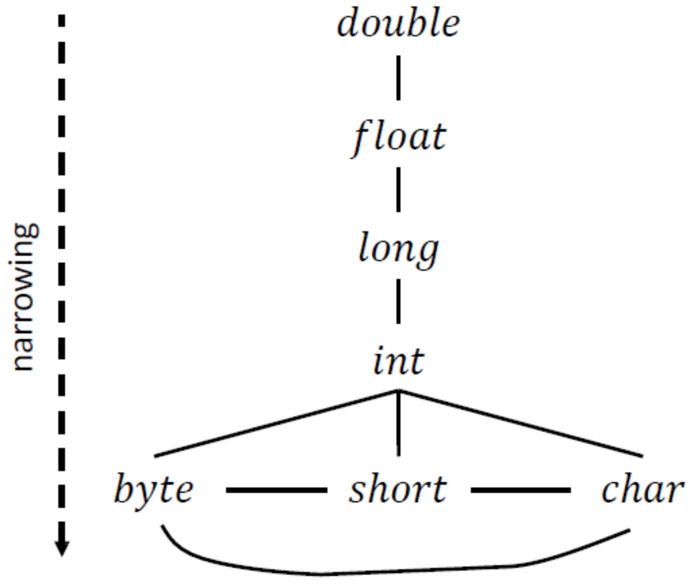
Compilers build trees to represent types

- Construct a tree for each type declaration and compare tree structures to test for equivalence

Type Conversion

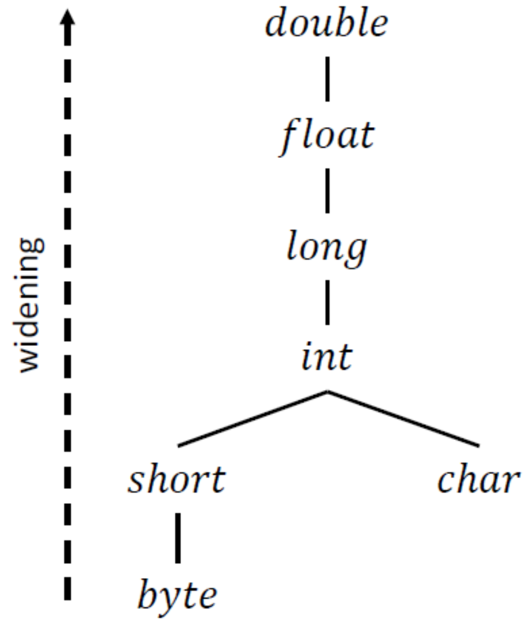
$E \rightarrow E_1 + E_2$	{ if ($E_1.type == integer$ and $E_2.type == integer$) $E.type = integer$ else if ($E_1.type == float$ and $E_2.type == integer$) $E.type = float$ …}





Type Conversion

$E \rightarrow E_1 + E_2$	{ if ($E_1.type == integer$ and $E_2.type == integer$) $E.type = integer$
	 else if ($E_1.type == float$ and $E_2.type == integer$) $E.type = float$
	 …}

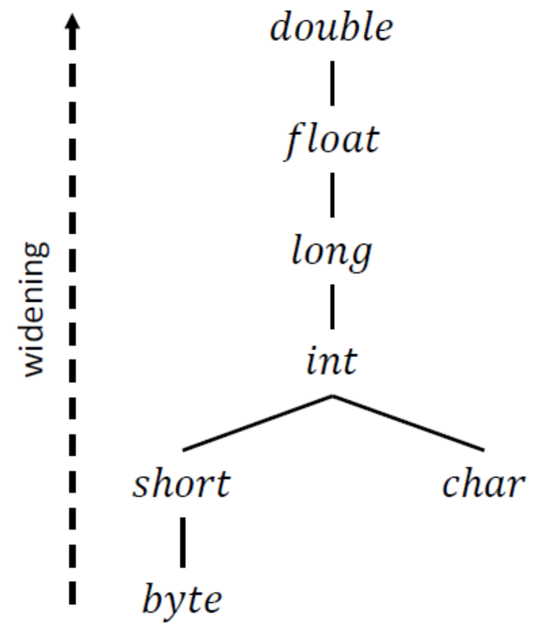


Assume two helper functions:

- $\max(t_1, t_2)$ – return the maximum (or least common ancestor) of the two types in the hierarchy
- $\text{widen}(a, t, w)$ – widen a value of type t at address a into a value of type w

Type Conversion

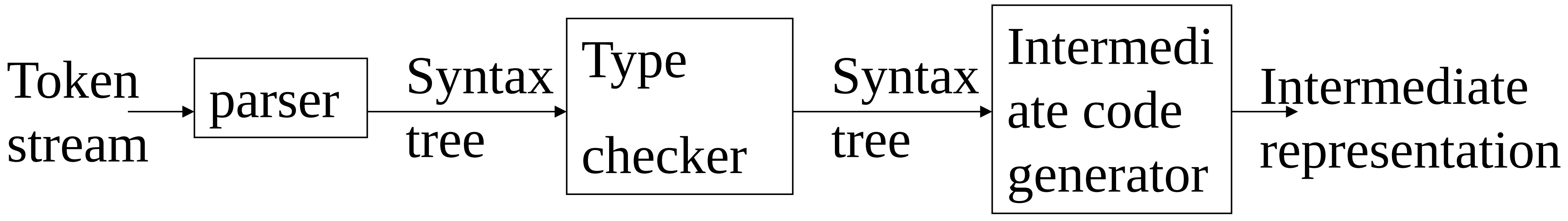
$E \rightarrow E_1 + E_2$	$\{\, E.type = \max(E_1.type, E_2.type);\ a_1 = widen(E_1.addr, E_1.type, E.type);$ $a_2 = widen(E_2.addr, E_2.type, E.type);$ $E.addr = new\ Temp();\ \ gen(E.addr = a_1\ "+"\ a_2); \}$



Assume two helper functions:

- $\max(t_1, t_2)$ - return the maximum (or least common ancestor) of the two types in the hierarchy
- $\text{widen}(a, t, w)$ – widen a value of type t at address a into a value of type w

Position of type checker



Notes: 1)A type checker verifies that the type of a construct matches that expected by its context.

2)Type information gathered by a type checker may be needed when code is generated.

The design of a type checker for a language is based on the information about the syntactic constructs in the language.

Each expression has a type associated with it.

Types of Type Checking

There are two kinds of type checking:

- Static Type Checking.
- Dynamic Type Checking.

Static Type Checking

- Static Type Checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variable is known at the compile time.
- Static Type-Checking is also used to determine the amount of memory needed to store the variable.

Examples of Static checks include:

Type-checks: A compiler should report an error if an operator is applied to an incompatible operand. For example, if an array variable and function variable are added together.

The flow of control checks: Statements that cause the flow of control to leave a construct must have someplace to which to transfer the flow of control. For example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement, an error occurs if such an enclosing statement does not exist.

Uniqueness checks: There are situations in which an object must be defined only once. For example, an identifier must be declared uniquely, labels in a case statement must be distinct, and else a statement in a scalar type may not be represented.

Name-related checks: Sometimes the same name may appear two or more times. For example in Ada, a loop may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

The Benefits of Static Type Checking

Runtime Error Protection.

It catches syntactic errors like spurious words or extra punctuation.

It catches wrong names like Math and Predefined Naming.

Detects incorrect argument types.

It catches the wrong number of arguments.

It catches wrong return types, like return “70”, from a function that’s declared to return an int.

Dynamic Type Checking

Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables.

For example, Python is a dynamically typed language. It means that the type of a variable is allowed to change over its lifetime. Other dynamically typed languages are -Perl, Ruby, PHP, Javascript etc.

Type systems

The type system is a set of rules for associating type expressions with varying parts of the program. A type checker implements a type system. Distinct compilers or processors of the system Language may employ different type systems.

Our project

Toy Language Types

```
Type ::= int | bool | string
               | Ref Type
              | void

Value ::= Number
  | True  |False
  | String
  | (value, Type)  //ref value
```

Toy Language Syntax

Stmt ::= Stmt;Stmt

| Id= Exp | Type Id

| print(Exp)| If Exp Then Stmt1 else Stm2

| Nop

| openRFile(Exp)

| readFile(Exp, id)

| closeRFile(Exp)

| new(Id, Exp)

| wH(Id, Exp)

| while Exp Stmt

| fork(Stmt)

Toy Language Syntax

```
Exp ::= Value
  | id
   | rH(Exp)
  | Exp1 + Exp2  | Exp1 - Exp2
   | Exp1 * Exp2 | Exp1 / Exp2
  | Exp1  and Exp2| Exp1 or Exp2
  | Exp1 < Exp2| Exp1 <= Exp2| Exp1 == Exp2
   | Exp1 != Exp2| Exp1  > Exp2 | Exp1 >= Exp2
```

Types Rules

<u>**Types**</u>

2 : int

id : type (given by the programmer)

<u>**Type rules**</u>

```
1: int     2:int
--------------------------
1 + 2 : int
```

is reading as:

-- 1+2 has type int IF 1 has type int and 2 has type int

or

-- IF 1 has type int and 2 has type int THEN 1+2 has type int

Types Rules

1: int   v:??

----

1 + v : int

G- type environment, defined as a list of pairs (id:type)

G|-1: int   G|-v:int

--------------------------   where G=[v:int]

G|-1 + v : int

Types Rules for values

G|- Number : int

G|-True:bool

G|-False:bool

G|-String:string

Types Rules for values

G|- val: int

G|-(val,type) : Ref type

Types Rules for expressions

(id:t) is in G

G|- id:t

G|- e1: int   G|-e2:int

G|-e1 + e2:int

the same rule for -,*,/

Types Rules for expressions

```
 G|- e1: bool    G|-e2:bool
---------------------------
G|-e1  and e2:bool
```

the same rule for or

```
G|- e1: int    G|-e2:int
---------------------------
G|-e1  < e2:bool
```

the same rule for <=,==,!=,>, >=

Types Rules for expressions

G|- e1 :Ref t1

G|- rH(e1): t1

Types Rules for statements

G|- s1:void,G1   G1|- s2:void,G2

G|- s1;s2: void,G2

G|- id:t1   G|- exp:t2 t1==t2

G|- id=exp: void,G

Types Rules for statements

G|- type id : void, G+[(id:type)]

G|- exp:t

G|- print(exp): void,G

Types Rules for statements

```
 G|- e : bool

G|- s1:void,G1

G|- s2:void,G2

---------------------------

G|- if e then s1 else s2 : void,G



-----------------

G|- nop:void, G
```

Types Rules for statements

G|- exp:string   G|-id:int

G|- readFile(exp,id):void,G

G|- exp:string

G|- openRFile(exp):void, G

Types Rules for statements

```
 G|- exp:string
--------------------------
G|- closeRFile(exp):void, G


G|- exp:t    G|- id:Ref t
-----------------------------
G|- new(id,exp):void, G
```

Types Rules for statements

```
 G|- exp:t    G|- id:Ref t
-----------------------------
G|- wH(id,exp):void, G


G|- exp:bool    G|- stmt:void,G1
-----------------------------------------
G|- while exp stmt:void, G
```

Types Rules for statements

```
  G|- stmt:void,G1
----------------------------------------
G|- fork(stmt):void, G
```