Type Checking

Nandini Mukherjee

Types and Typesystems

Types:

- collection of values from a "domain" (the denotational approach)
- internal structure of a bunch of data, described down to the level of a small set of fundamental types (the structural approach)
- equivalence class of objects (the implementor's approach)
- collection of well-defined operations that can be applied to objects of that type (the abstraction approach)

- **A Typesystem** is a set of rules which assign types to expressions, statements, and thus the entire program
  - what operations are valid for which types
  - concise formalization of the checking rules
  - specified as rules on the structure of expressions
  - language specific

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
  - C, C++, Java
- Dynamically typed language: type of an expression determined at run time
  - Python, Ruby, Javascript
- Untyped language: no typechecking, e.g., assembly

Static Type Checking

- Achieved typically through a type system
- Interfaces between different parts of a program are defined
- It is checked that parts have been connected in a consistent way
- Thereby, reduces possibilities for bugs in programs

Why Static Typing?

- Compiler can reason more effectively
- Allows more efficient code: No need to check for unsupported operations
- Allows error detection by compiler – *type error*
- Documentation of code can be done
- However,
  - requires at least some *type declarations*
  - type declarations often can be inferred

Dynamic checks

- Array index out of bounds
- null in Java, null pointers in C
- Inter-module type checking in Java
- Sometimes can be eliminated through static analysis (but usually harder than type checking)

Dynamic Type Checking

- Each run-time object uses a tag with type information
- This information is used to check for type errors
- Many statically typed languages implement some form of dynamic type checking
  - Because some useful features and non-trivial properties are difficult to verify statically
  - e.g. downcasting – A dynamic check is needed to verify that the operation is safe or not during downcasting
  - If not safe, a ClassCastException is thrown
- Dynamic type checking may cause a program to fail at runtime
- In some programming languages, it is possible to anticipate and recover from these failures
- In others, type-checking errors are considered to be fatal.

```java
// Parent class
class Parent {
        String name;
        // A method which prints the
        // signature of the parent class
        void method() {
        System.out.println("Method from Parent");}
}
// Child class
        class Child extends Parent {
        int id;
        // Overriding the parent method to print
        //the signature of the child class
        @Override void method() {
        System.out.println("Method from Child");}
}
```

```java
public class GFG {
// Driver code
public static void main(String[] args) {
        // Upcasting
        Parent p = new Child();
        p.name = “Hello World";
        //Printing the parentclass name
        System.out.println(p.name);
        //parent class method is overridden method
        p.method();
        // Implicit Downcasting will give compile time
        // error
        // Child c = new Parent(); - > compile time error
        // Downcasting Explicitly
        Child c = (Child)p;
        c.id = 1;
        System.out.println(c.name);
        System.out.println(c.id);
        c.method(); }
}
```

Output

Hello World

Method from Child

Hello World

1

Method from Child

Parent p = new Child();

- Upcasting will be done internally
- the object is allowed to access only parent class members and specific child class members (overridden methods, etc.), but not all members (p.id is not accessible)

Child c = (Child)p;

- Downcasting has to be done externally
- a child object can acquire the properties of the parent object.

Sound Type System

- If an expression is assigned with type *t*, and it evaluates to a value *v*, then *v* is in the set of values defined by *t*
- In other words, dynamic type of expression (at runtime) is the static type of the expression (derived at compile time)
- SML, OCAML and Ada have sound type systems
- Most implementations of C and C++ do not

Strongly Typed Language

- Strong vs. weak typing
  - strong: guarantees no illegal operations performed
  - weak: can’t make guarantees
- When no application of an operator to arguments can lead to a run-time type error, language is strongly typed
- strongly typed is not same as statically typed
- C++ claim to be “strongly typed”, but
  - Union types allow creating a value of one type and using it at another
  - Type coercions may cause unexpected 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

Type Expressions

- Type expressions are used in declarations and type casts to define or refer to a type
  - *Primitive types*, such as int and float
  - *Type constructors*, such as pointer-to, array-of, records and classes, templates, and functions
  - *Type names*, such as typedefs in C
- Some languages allow type aliases
  - E.g. in C: `typedef int int_array[ ];`
  - int_array is type expression denoting same type as int [ ]

Type Expressions

- Different languages have various kinds of array types
  - without bounds: array(T): *C, Java:* *T[ ]*
  - size: array(T, L) (may be indexed 0..L-1): *C:* *T[L]*
  - upper and lower bounds: array(T,L,U): *Pascal, Modula-3: indexed L..U*
  - Multi-dimensional arrays (FORTRAN)

Records or Structures

- More complex type constructor
  - Has form {id1: T1, id2: T2, …} for some ids and types Ti
- Supports access operations on each field, with corresponding type
- E.g. C: *struct { int a; float b; }*

  corresponds to type {a: int, b: float}

Functions

- Some languages have first-class function types
  - C, Modula-3, Pascal have first-class function types
  - Java is not considered to have first-class function types
- Function value can be invoked with some argument expressions with types Ti, returns return type Tr.

C: *int f(float x, float y)*

Type Equivalence

Two aspects of type equivalence

- Name equivalence: Each distinct type name is a distinct type.
  - Used by Pascal
- Structural Equivalence: two types are identical if they have the same structure
  - Used in C, Java

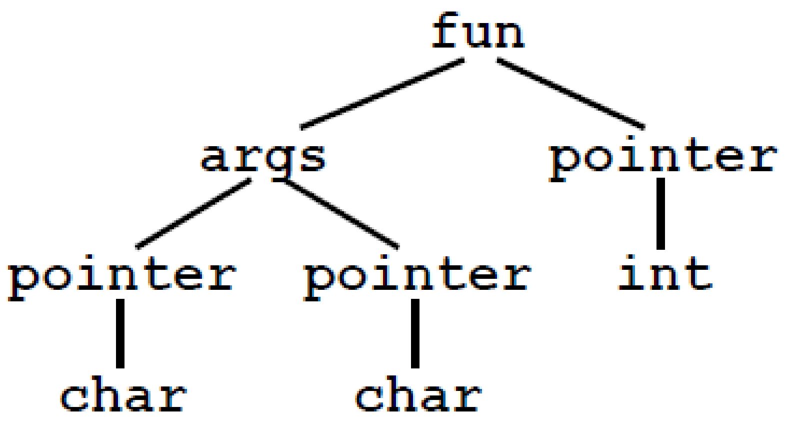
```
typedef node* link;
link next;
link last;
node* p;
node* q;
```

Using structural equivalence:

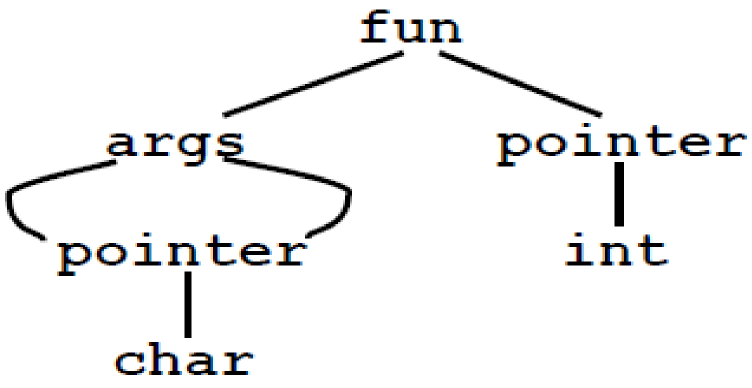
```
p = q = next = last
```

Representing Types

int *f(char*,char*)



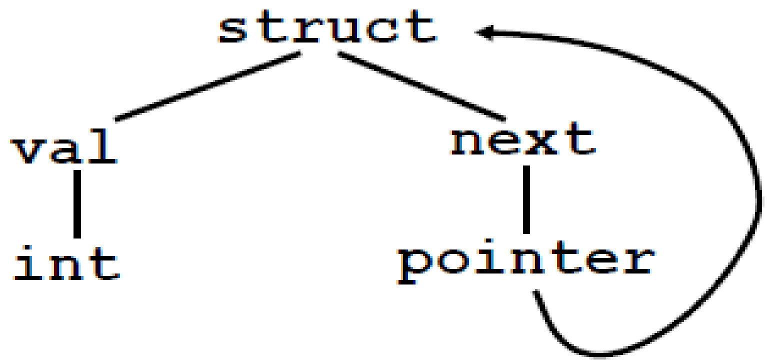
Tree form



DAG form

Cyclic Graph Representations

```
struct Node
{
int val;
struct Node *next;
};
```

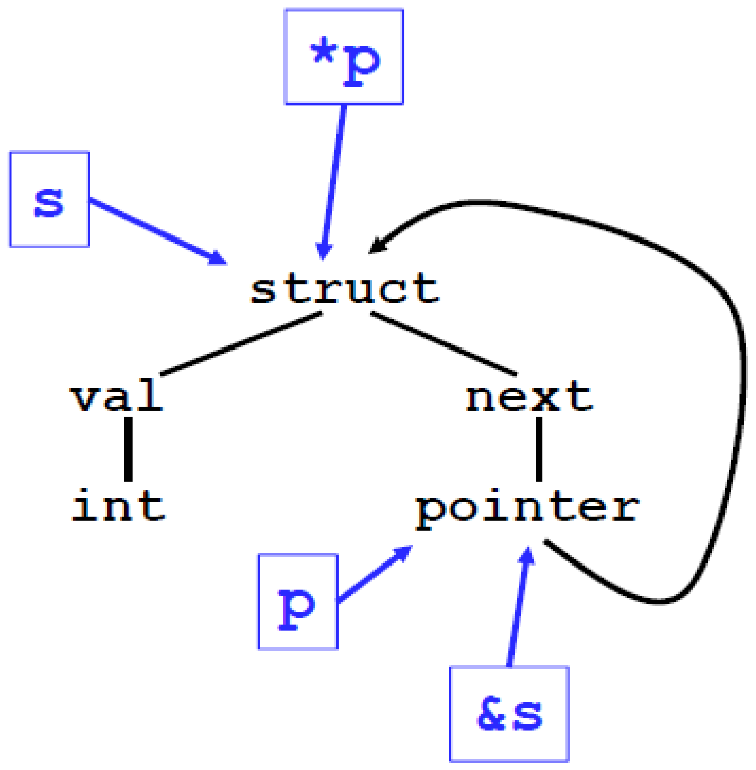


Cyclic graph

Structural Equivalence

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node {
  int val;
  struct Node *next;
};
struct Node s, *p;
… p = &s; // OK
… *p = s; // OK
```



Structural Equivalence

- if atomic types, then obvious
- if type constructors:
  - same constructor
  - recursively, equivalent arguments to constructor
- implement with recursive implementation of equals
- e.g. atomic types, array types, record types in ML

Type Conversions and Coercions

In Java

- Explicitly converts an object of type double to one of type int
  - can be represented as unary operator
  - typecheck, code generation done normally
- Implicitly coerce an object of type int to one of type double
  - compiler must insert unary conversion operators, based on result of type checking

Type Casting

- C and Java can explicitly cast an object of one type to another
- Sometimes casting means a conversion (casts between numeric types)
- Sometimes casting means just a change of static type without doing any computation (casts between pointer types or pointer and numeric types)
- In C, safety/correctness of casts not checked
  - allows writing low-level code that is type-unsafe
  - more often used to work around limitations in C’s static type system
- In Java downcasts from superclass to subclass include run-time type check to preserve type safety
  - static typechecker allows type casting
  - codegeneration introduces run-time check
  - Purpose of dynamic type checking

Constructing Type Graphs

- Construct over AST (or during parse)

type	→	**int**	$$ = getIntType();
	|	**bool**	$$ = getBoolType();
	|	***** type	$$ = makePtrType($2);
	|	type **[** num **]**	$$ = makeArrayType($1, $3);
typedef	→	typedef type id	install($3,$2);

Type Checking

- Type checking ensures that operations are applied to the right number of arguments of the right types
  - i.e. same type as was specified, or
  - there is a predefined implicit coercion that will be applied
- Used to resolve overloaded operations
- May be done statically at compile time or dynamically at run time
- Dynamically typed languages do only dynamic type checking
- Statically typed languages can do **most** type checking statically

Dynamic Type Checking

- Performed at run-time before each operation is applied
- Types of variables and operations are left unspecified until run-time
  - Same variable may be used at different types
- Data object must contain type information
- Errors are not detected until violation
- May introduce extra overhead at runtime.
  - Can make code hard to read

Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time
- Typically places restrictions on languages
  - Issues with garbage collection
  - References instead of pointers
  - All variables are initialized when created
  - Variable only used at one type
  - Union types are allowed, but effectively introduce dynamic type checks

Type Inference

- *Type inference:*
- Assigning a type to an expression from the program context of the expression
  - Fully static type inference first introduced by Robin Miller in ML
  - Haskle, OCAML, SML all use type inference
  - Records are a problem for type inference

Generic Type Checking Algorithm

- To do semantic analysis & checking on a program,
  - recursively type check each node in the program's AST in the context of the symbol table for its enclosing scope
    - going down, create any nested symbol tables & context needed
    - recursively type check child subtrees
    - on the way back, check that the children are legal in the context of their parents
- Each AST node class defines its own type check method, which fills in the specifics of this recursive algorithm
- Generally:
  - declaration AST nodes add bindings to the current symbol table
  - statement AST nodes check their subtrees
  - expression AST nodes check their subtrees and return a result type

Limitations of type checking

- Can still have runtime errors:
  - division by zero
  - exceptions
- Static type analysis has to be conservative,
  - thus some “correct” programs may be rejected.