

CS 335: Type Systems

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II



Type Error in Python

```
def add(x):  
    return x + 1  
  
class A(object):  
    pass  
  
a = A()  
add(a)
```

```
> python3 --version  
Python 3.10.12  
> python3 type.py  
Traceback (most recent call last):  
  File "/home/swarnendu/Desktop/type.py", line 8, in <module>  
    add(a)  
  File "/home/swarnendu/Desktop/type.py", line 2, in add  
    return x + 1  
TypeError: unsupported operand type(s) for +: 'A' and 'int'
```

Compilers help detect type errors

What is a Type?

Definition

Type is a set of values and operations allowed on those values

- Integer is any whole number in the range $-2^{31} \leq i < 2^{31}$ and examples of allowed operations are $+$, $-$, \times , and \div
- Booleans have true and false values and examples operations are $\&\&$, $\|\|$, and $!$

Abstraction-based A type is an interface consisting of a set of operations with well-defined and mutually consistent semantics

Structural A type is either from a collection of built-in types or a composite type created by applying a type constructor to built-in types

Denotational

- A type is a set of values
- A value has a given type if it belongs to the set

What is a Type?

Pascal

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

C

The result of the unary `&` operator is a pointer to the object referred to by the operand. If the type of operand is X , the type of the result is a pointer to X .

The type of a language construct is denoted by a type expression (e.g., basic types like `int` and `float`)

Type System

Definition

The set of types and rules to associate type expressions to different parts of a program (e.g., variables, expressions, and functions) are collectively called a **type system**

- Type systems include rules for type inference, type equivalence, and type compatibility
 - ▶ Type inference defines the type of an expression based on the types of its constituent parts or the surrounding context
 - ▶ Type equivalence determines when types of two values are the same
 - ▶ Type compatibility determines when a value of a given type can be used in a given context
- Goal is to reduce sources of bugs due to type errors

- Different type systems may be used by different compiler implementations for the same language
 - ▶ Pascal language specification includes the index set of arrays in the type information of a function, but compiler implementations allow the index set to be unspecified

Type Checking

- Ensure that valid operations are invoked on variables and expressions
 - ▶ E.g., && operator in Java expects two operands of type boolean
- Includes both type inferencing and identifying type-related errors
 - ▶ A **type error** (or type clash) occurs when we attempt an operation on a value for which that operation is not defined
 - ▶ Can catch errors, so needs to have a notion for error recovery
 - ▶ Run-time errors (e.g., arithmetic overflow) is outside the scope of type systems
- A type checker implements a type system

Catching Type Errors

- Impossible to build a type checker that can **predict** which programs will result in type errors

```
class A {  
    int add(int x) {  
        return x + 1;  
    }  
    public static void main(String args[]) {  
        A a = new A();  
        if (false) { add(a); }  
    }  
}
```

- Type checkers make a **conservative** approximation of what will happen during execution
 - ▶ Raises error for anything that might cause a type error

```
> javac type.java  
type.java:8: error: incompatible types: A cannot be converted to int  
if (false) { add(a); }  
                ^
```

Note: Some messages have been simplified; recompile with `-Xdiags:verbose` to get full output
1 error

Type Safety

- A program is **type-safe** if it is known to be free of type errors
- A language is type-safe if the only operations that can be performed on data in the language are those allowed by the type of data
 - ▶ All legal programs in that language are type-safe
- Type-safe languages do not allow operations or conversions that violate the rules of the type system
- Java, Smalltalk, Scheme, Haskell, Ruby, and Ada are examples of type-safe languages, while Fortran and C are not type-safe

Categories of Type Systems

Strongly-typed Every expression can be assigned an unambiguous type

Weakly-typed Allows a value of one type to be treated as another

- Errors may go undetected at compile time and even at run time

Untyped Allows any operation to be performed on any data

- No type checking is done (e.g., Assembly, Tcl, and BCPL)

What is the difference between a strongly typed language and a statically typed language?

Categories of Type Systems

Statically-typed Every expression can be typed during compilation (e.g., C, C++, Java, and Rust)

Dynamically-typed Types are associated with run-time values rather than expressions (e.g., Lisp, Perl, Python, Javascript, and Ruby)

- Type errors cannot be detected until the code is executed

```
>>> if False:
...     1 + "two"
... else:
...     1 + 2
...
3
>>> 1 + "two"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

What is the difference between a strongly typed language and a statically typed language?

Static vs Dynamic Typing

Static Typing

- Can find errors at compile time
- Low cost of fixing bugs
- Improved reliability and performance of compiled code
- Effectiveness depends on the strength of the type system

Dynamic Typing

- Allows fast compilation
- Type of a variable can depend on run time information
- Can load new code dynamically
- Allows constructs that static checkers would reject

Categories of Type Systems

Static type systems can be explicit or implicit depending on whether the types are explicitly written

- **Manifest** (or **explicit**) typing requires explicitly identifying the type of a variable during declaration (e.g., Pascal and Java)

- Type is deduced from context in **latent** (or **implicit**) typing (e.g., Standard ML and OCaml)

```
int main() {  
    float x = 0.0;  
    int y = 0;  
    ...  
}
```

```
(* Standard ML*)  
let val s = "Test"  
    val x = 0.0  
    val y = 0  
in print "Hello, World!\n"  
end
```

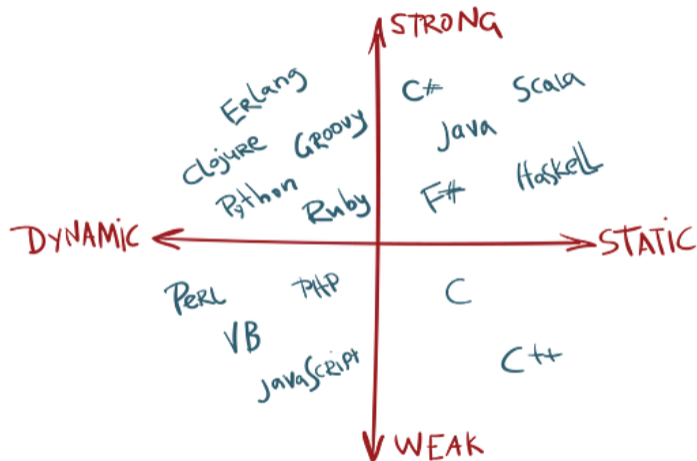
Categorization of Programming Languages

	Statically-Typed	Dynamically-Typed
Strongly-Typed	ML, Haskell, Java, Pascal	Lisp, Scheme
Weakly-Typed	C, C++	Perl, PHP

- C is weakly and statically typed
- C++ is statically typed with optional dynamic type casting
- Some languages allow both static and dynamic typing
 - ▶ Java is both statically and dynamically typed (allows downcasting to subtypes)
- Python is strongly and dynamically typed

Is Python strongly typed?

Categorization of Programming Languages



Magic lies here - Statically vs Dynamically Typed Languages

Comparison of programming languages by type system

More on Type Checking

- All checking can be implemented dynamically
- A **sound** type system ensures that the type of the value computed from an expression matches the expression's static type, and thus avoids the need for dynamic checking
- A compiler can implement a statically typed language with dynamic checking

Gradual Typing

- Allows parts of a program to be either dynamically typed or statically typed
- Programmer controls the typing with annotations
 - ▶ Unannotated variables have an unknown type, check type at run time
 - ▶ Static type checker considers every type to be compatible with unknown
 - ▶ E.g., Typescript and CPerl

```
# Type Hints in Python v3.5+ (PEP 484)  
def sum(num1: int, num2: int) -> int:  
    return num1 + num2  
  
print(sum(2, 3))  
print(sum(1, 'Hello World!'))
```

No type checking will happen at run time

Are These Types Same?

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

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

Nominal and Structural Typing

- **Nominal** typing requires that the object is exactly of the given type (by name) or is a subtype of that type (e.g., C++, Java, and Swift)
- **Structural** typing requires that an object supports a **given set** of operations even if some of them may not be used (e.g, Ocaml, Haxe, and Haskell)

```
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!
```

Nominal and Structural Typing

Nominal Typing

```
function greet(person) {  
  if (!(person instanceof Person))  
    throw TypeError;  
  alert("Hello, " + person.Name);  
}
```

Structural Typing

```
function greet(person) {  
  if (!(typeof(person.Name) == string  
    && typeof(person.Age) == number))  
    throw TypeError;  
  alert("Hello, " + person.Name);  
}
```

Is it possible to have a dynamically typed language without duck typing?

Duck Typing

An object's validity is determined by the presence of certain methods and properties, rather than the type of the object itself

```
class Duck:
    def fly(self):
        print('Duck flying')
class Sparrow:
    def fly(self):
        print('Sparrow flying')
class Whale:
    def swim(self):
        print('Whale swimming')

for animal in Duck(), Sparrow(), Whale():
    animal.fly()
```

If it walks like a duck and it quacks like a duck, then it must be a duck

TypeScript Example

```
interface Person {  
  Name : string;  
  Age : number;  
}  
function greet(person : Person) {  
  console.log('Hello, ' + person.Name);  
}  
greet({ Name: 'svick' });
```

no Age property

```
function greet(person : Person) {  
  console.log('Hello, ' + person.Name);  
}  
greet({ Name: 'svick' });
```

- Compilation error implies TypeScript uses static structural typing
- Code still compiles to JavaScript
 - ▶ Implies TypeScript makes use of dynamic duck typing

Is it possible to have a dynamically typed language without duck typing?

Benefits of Types

Usefulness of Types

Type systems help specify precise program behavior

- Hardware does not distinguish the interpretation of a sequence of bits
- Assigning type to a data variable, called typing, gives meaning to a sequence of bits

Abstraction Enables thinking in terms of primitive or composite data structures

Safety Disallows meaningless computations, limits the set of operations in a semantically valid program

Optimizations Static type checking may allow a compiler to use specialized instructions for data types

Documentation Clarifies the intent of the programmer on the nature of the computation, helps reduce bugs

Ensure Run-time Safety

- Well-designed type system helps the compiler detect and avoid run-time errors by identifying misinterpretations of data values

	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	illegal
complex	complex	complex	illegal	complex

Result Types for Addition in Fortran 77

Enhanced Expressiveness

Strong type systems can support other features

- Operator overloading gives **context-dependent meaning** to an operator
 - ▶ A symbol that can represent different operations in different contexts is overloaded
- Many operators are overloaded in typed languages
- In untyped languages, lexically different operators are required

Strong type systems allow generating efficient code

- Perform compile-time optimizations, no run-time checks are required
- Otherwise, the compiler needs to maintain type metadata along with the value

Implementing Addition

Strongly-Typed Systems

integer \leftarrow integer + integer

```
add %ra %rb %ra+b
```

double \leftarrow integer + double

```
movss (double_var) %xmm0  
cvtsi2ss %ri %xmm1  
addss %xmm1 %xmm0
```

Weakly-Typed Systems

```
if type_md(a) == int  
  if type_md(b) == int  
    value(c) = value(a) + value(b)  
    type_md(c) = int  
  else if type_md(b) == float  
    temp = convert_to_float(a)  
    value(c) = temp + value(b)  
    type_md(c) = float  
  else ...  
else ...
```

Classification of Types

Components of a Type System

- (i) Basic (or built-in) types
- (ii) Rules for constructing new types from basic types
- (iii) Method for checking equivalence of two types
- (iv) Rules to infer the type of a source language expression

Base Types

- Modern languages include types for operating on numbers, characters, and Booleans
 - ▶ Influenced by operations supported by the hardware
- Individual languages may add additional types
 - ▶ Exact definitions and types vary across languages
 - ▶ E.g., 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

- Programs often involve ADT concepts like graphs, trees, and stacks
 - ▶ Each component of an ADT has its own type
- Constructed (also called non-scalar) types are created by applying a type constructor to one or more base types
 - ▶ Examples are arrays, strings, enums, structures, and unions
 - ▶ Lists in Lisp are constructed type: A list is either nil or (cons first rest)
- Constructed types can allow high-level operations (e.g., assign one structure variable to another variable)

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

Type of Node may be $(\text{Node}^*) \times \text{int}$

```
union Data {  
    int i;  
    float f;  
    char str[16];  
};
```

Type of Data may be $\text{int} \cup \text{float} \cup \text{char}[]$

Array 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 (a set of integers)

```
int A[10];
```

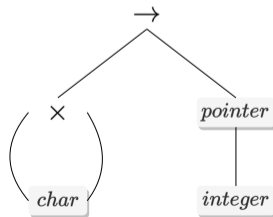
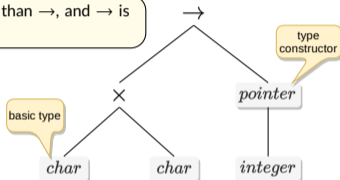
- Array A can have type expression $array(0 \dots 9, integer)$
 - ▶ C uses equivalent of int^* as the array type
- If T_1 and T_2 are type expressions, then the Cartesian product $T_1 \times T_2$ is a type expression

Function Type Constructors

- Function maps domain set to a range set denoted by type expression $D \rightarrow R$
- Type of function `int* f1(char a, char b);` can be denoted by `char × char → int*`
- Type signature is a specification of the types of the formal parameters and return value(s) of a function

Tree and DAG representation of the type expression `char × char → pointer(integer)`

× has higher priority than →, and → is right-associative



Pointer Type Constructors

- If T is a type expression, then $pointer(T)$ is a type expression denoting type pointer to an object of type T
- Type safety with pointers assumes addresses correspond to typed objects
- Ability to construct new pointers complicates reasoning about pointer-based computations
 - ▶ Some languages allow manipulating pointers
 - ▶ Autoincrement and autodecrement construct new pointers

Polymorphism

Ad hoc and Coercion Polymorphism

Polymorphism means using 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 polymorphism refers to functions of the same name whose **behavior depends on the type** of arguments
 - ▶ E.g., function and operator overloading
 - Coercion polymorphism occurs when **primitives or objects are cast** to other types

```
String frts = "Apple" + "Orange";  
int a = b + c;
```

Explicitly specifies the set of types at compile time

```
int(43.2)  
double dnum = Double.valueOf(inum);
```

Parametric Polymorphism

- Code takes **type or set of types as a 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 (in Java) or templates (in C++)
 - ▶ 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 to **access derived class objects through base class pointers**
 - ▶ 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 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 by method overriding

- Usually used with subtype polymorphism

Type Equivalence

Are These Definitions The Same?

- Should the reversal of the order of the fields change type?
 - ▶ Some languages (e.g., ML) say no, and 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 [0...9] of char;
```

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

- In nominal equivalence, two type expressions are the same if they have the same name (e.g., C++, Java, and Swift)
- In structural equivalence (e.g., OCaml and Haskell), two type expressions are equivalent if
 - (i) Either both are the same basic types, or
 - (ii) Are formed by applying the same type constructor to equivalent types

Nominal vs Structural Equivalence

Nominal Equivalence

- Equivalent if the type names are the same
 - ▶ Identical names can be intentional
 - ▶ Can avoid unintentional clashes
 - ▶ Difficult to scale for large projects

Structural Equivalence

- Equivalent only if the types have the same structure
 - ▶ Assumes interchangeable objects can be used in place of one other
 - ▶ Problematic if values have special meanings

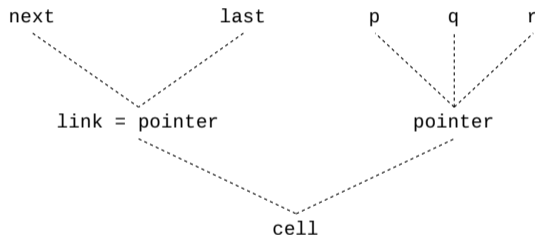
Compilers build trees to represent types

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

Type Graph

Two type expressions are equivalent if they are represented by the same node in the type graph

```
type link = ↑ cell;  
var next : link;  
    last : link;  
    p : ↑ cell;  
    q, r : ↑ cell;
```



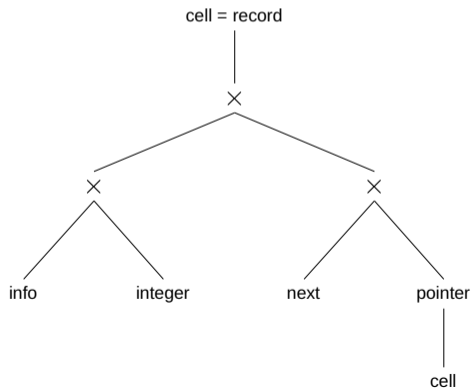
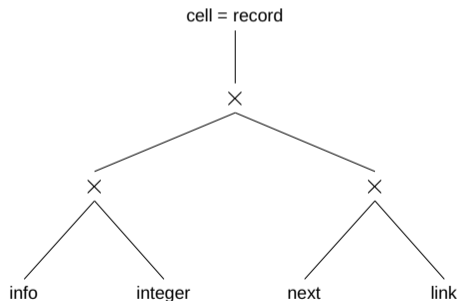
- Under nominal equivalence, `next` and `last`, and `p`, `q`, and `r` are of the same type
- Under structural equivalence, all the variables are of the same type
- An alternate policy is to assign implicit type names every time a type name appears in declarations
 - ▶ Type expressions of `p` and `q` will then have different implicit names

Testing for Structural Equivalence

```
bool struc_equiv(type s, type t) {  
  if s and t are the same basic type then  
    return true  
  else if s = array(s1,s2) and t = array(t1,t2) then  
    return struc_equiv(s1, t1) and struc_equiv(s2, t2)  
  else if s = s1 × s2 and t = t1 × t2 then  
    return struc_equiv(s1, t1) and struc_equiv(s2, t2)  
  else if s = pointer(s1) and t = pointer(t1) then  
    return struc_equiv(s1, t1)  
  else if s = s1 → s2 and t = t1 → t2 then  
    return struc_equiv(s1, t1) and struc_equiv(s2, t2)  
  else  
    return false  
}
```

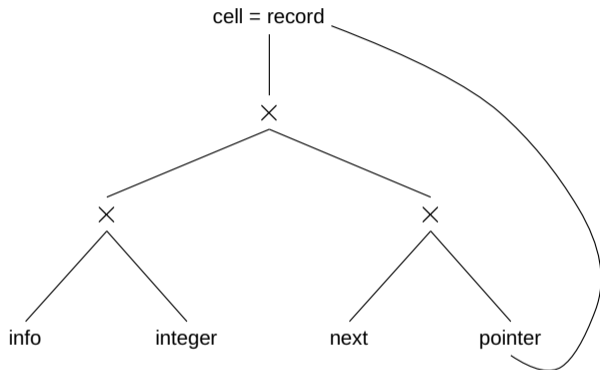
Representing Recursively-Defined Types

```
type link = ↑ cell;  
cell = record  
  info : integer;  
  next : link  
end;
```



Cycles in Representations of Types

```
type link = ↑ cell;  
cell = record  
  info : integer;  
  next : link  
end;
```



Type Equivalence

- C uses structural equivalence for scalar types and uses nominal equivalence for structs
- Language in which **aliased types are distinct** is said to have **strict name** equivalence
- Loose name equivalence implies aliased types are considered equivalent

```
int *p, *q, *r;  
typedef int * pint;  
pint start, end;
```

Variable	Type Expression
p	pointer(int)
q	pointer(int)
r	pointer(int)
start	pint
end	pint

Efficient Encoding of Type Expressions

- Bit vectors can be used to encode type expressions more efficiently than graph representations
 - ▶ *pointer*(*t*) denotes a pointer to type *t*
 - ▶ *array*(*t*) denotes an array of elements of type *t*
 - ▶ *func*(*t*) denotes a function that returns an object of type *t*

Type Constructor	Encoding	Basic Type	Encoding
<i>pointer</i>	01	boolean	0000
<i>array</i>	10	char	0001
<i>func</i>	11	integer	0010
		real	0011

Type Expression	Encoding
char	000000 0001
<i>func</i> (char)	000011 0001
<i>pointer</i> (<i>func</i> (char))	000111 0001
<i>array</i> (<i>pointer</i> (<i>func</i> (char)))	100111 0001

six bits are used because there are three type constructors

Efficient Encoding of Type Expressions

- Bit vectors can be used to encode type expressions more efficiently than graph representations
 - ▶ *pointer*(*t*) denotes a pointer to type *t*
 - ▶ *array*(*t*) denotes an array of elements of type *t*
 - ▶ *func*(*t*) denotes a function that returns an object of type *t*

Type Constructor	Encoding	Basic Type	Encoding	Type Expression	Encoding
<i>pointer</i>	11	integer	0010	<i>pointer</i> (<i>func</i> (char))	0001110001
<i>array</i>	10	real	0011	<i>array</i> (<i>pointer</i> (<i>func</i> (char)))	1001110001
<i>func</i>	01				

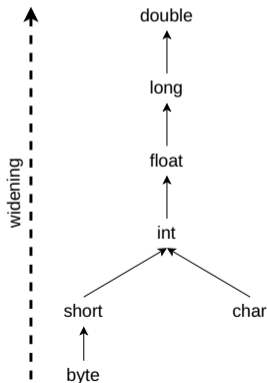
Encoding saves space and also tracks the order of the type constructors in type expressions

Type Inference Rules

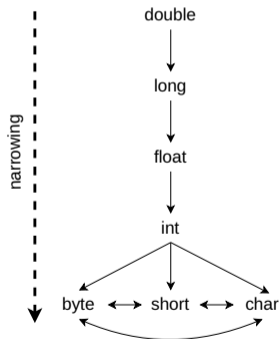
- Specifies, for each operator, the mapping between the operand types and the result type
 - ▶ Type of the LHS of an assignment must be the same as the RHS
 - ▶ In Java, for example, adding two integer types of different precision produces a result of the more precise type
- Some languages require the compiler to perform implicit conversions
 - ▶ Internal representations of integers and floats are different in a computer
 - ▶ Recognize mixed-type expressions and insert appropriate conversions
 - ▶ Implicit type conversion done by the compiler is called **type coercion**
 - ▶ It is limited to the situations where no information is lost

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** $E_2.type == integer$) $E.type = float$
 ...

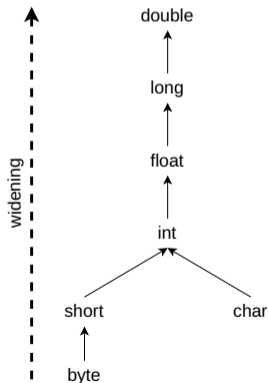


according to precision rules in Java



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

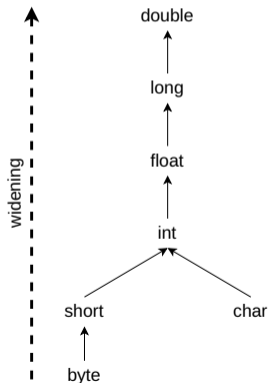
`max(t_1, t_2)` return the maximum (or least common ancestor) of the two types in the hierarchy

`widen(a, t, w)` widen a value of type t at address a into a value of type w

```
Addr widen(Addr a, Type t, Type w) {  
  if (t == w) return a;  
  else if (t == integer and w == float) {  
    temp = new Temp();  
    gen(temp '=' '(float)' a);  
    return temp;  
  } else { /* throw error; */  
}
```

Type Conversion

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



- 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

Type Checking

Type Checking Rules for Coercion from Integer to Real

Production	Semantic Actions
$E \rightarrow \text{num}$	$\{E.type = integer\}$
$E \rightarrow \text{num.num}$	$\{E.type = real\}$
$E \rightarrow \text{id}$	$\{E.type = lookup(\text{id.entry})\}$
$E \rightarrow E_1 \text{ op } E_2$	$\{ \text{if } (E_1.type == integer \text{ and } E_2.type == integer) E.type = integer$ $\text{else if } (E_1.type == integer \text{ and } E_2.type == real) E.type = real$ $\text{else if } (E_1.type == real \text{ and } E_2.type == integer) E.type = real$ $\text{else if } (E_1.type == real \text{ and } E_2.type == real) E.type = real \}$

Implicit conversion of constants at compile time
can reduce run time

Type Checking of Expressions

- **Idea:** build a parse tree, assign a type to each leaf element, assign a type to each internal node with a postorder walk
- Types should be matched for all function calls from within an expression
 - ▶ Possible ideas
 - (i) Require the complete source code
 - (ii) Make it mandatory to provide type signatures of functions as function prototype
 - (iii) Defer type checking until linking or run time

Type Synthesis for Overloaded Functions

- Suppose f is an overloaded function
- f can have type $s_i \rightarrow t_i$ for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$
- x has type s_k for some $1 \leq k \leq n \implies$ Expression $f(x)$ has type t_k

Specification of a Simple Type Checker

- Consider a language where each identifier must be declared before use
- **Goal:** Design a type checker that can handle statements, functions, arrays, and pointers

$$P \rightarrow D; E$$
$$D \rightarrow D; D \mid \mathbf{id} : T$$
$$T \rightarrow \mathbf{char} \mid \mathbf{integer} \mid \mathbf{array}[\mathbf{num}] \text{ of } T \mid \uparrow T$$
$$E \rightarrow \mathbf{literal} \mid \mathbf{num} \mid \mathbf{id} \mid E \text{ mod } E \mid E[E] \mid \uparrow E$$

Example string: `key:integer; key mod 1999`

SDT for Updating Types in Symbol Table

Production	Semantic Actions
$P \rightarrow D; E$	
$D \rightarrow D; D$	
$D \rightarrow \mathbf{id} : T$	$\{addtype(\mathbf{id}.entry, T.type)\}$
$T \rightarrow \mathbf{char}$	$\{T.type = char\}$
$T \rightarrow \mathbf{integer}$	$\{T.type = integer\}$
$T \rightarrow \mathbf{array}[\mathbf{num}] \text{ of } T_1$	$\{T.type = array(1 \dots \mathbf{num}.val, T_1.type)\}$
$T \rightarrow \uparrow T_1$	$\{T.type = pointer(T_1.type)\}$

Type Checking of Expressions

Production	Semantic Actions
$E \rightarrow \text{literal}$	$\{E.type = \text{char}\}$
$E \rightarrow \text{num}$	$\{E.type = \text{integer}\}$
$E \rightarrow \text{id}$	$\{E.type = \text{lookup}(\text{id.entry})\}$
$E \rightarrow E_1 \text{ mod } E_2$	$\{\text{if } (E_1.type == \text{integer} \text{ and } E_2.type == \text{integer}) \text{ then } E.type = \text{integer} \\ \text{else } E.type = \text{type_error}\}$
$E \rightarrow E_1[E_2]$	$\{\text{if } (E_2.type == \text{integer} \text{ and } E_1.type == \text{array}(s, t)) \text{ then } E.type = t \\ \text{else } E.type = \text{type_error}\}$
$E \rightarrow \uparrow E_1$	$\{\text{if } (E_1.type == \text{pointer}(t)) \text{ then } E.type = t \\ \text{else } E.type = \text{type_error}\}$

Type Checking of Statements

Statements do not have values, use the special basic type void

Production	Semantic Actions
$S \rightarrow \mathbf{id} = E$	{if (<i>id.type</i> == <i>E.type</i>) then <i>S.type</i> = void else <i>S.type</i> = type_error}
$S \rightarrow \mathbf{if} E \mathbf{then} S_1$	{if (<i>E.type</i> == <i>boolean</i>) then <i>S.type</i> = <i>S₁.type</i> else <i>S.type</i> = type_error}
$S \rightarrow \mathbf{while} E \mathbf{do} S_1$	{if (<i>E.type</i> == <i>boolean</i>) then <i>S.type</i> = <i>S₁.type</i> else <i>S.type</i> = type_error}
$S \rightarrow S_1; S_2$	{if (<i>S₁.type</i> == <i>void</i> and <i>S₂.type</i> == <i>void</i>) then <i>S.type</i> = <i>void</i> else <i>S.type</i> = type_error}

Type Checking of Functions

`int f(double x, char y) \implies f: double \times char \rightarrow int`

Production	Semantic Actions
$E \rightarrow E_1(E_2)$	{ if ($E_2.type == s$ and $E_1.type == s \rightarrow t$) then $E.type = t$ else $E.type = type_error$ }

Storage Layout for Local Variables

Production

$T \rightarrow BC$

$B \rightarrow \mathbf{int}$

$B \rightarrow \mathbf{float}$

$C \rightarrow \epsilon$

$C \rightarrow [num]C_1$

Determine the amount of allocation
(in bytes) in a declaration

Computing Types and Their Widths

Production

$T \rightarrow BC$

$B \rightarrow \mathbf{int}$

$B \rightarrow \mathbf{float}$

$C \rightarrow \epsilon$

$C \rightarrow [\mathit{num}]C_1$

Production

$T \rightarrow B \{t = B.type; w = B.width; \}$

$C \{T.type = C.type; T.width = C.width; \}$

$B \rightarrow \mathbf{int} \{B.type = \mathit{integer}; B.width = 4; \}$

$B \rightarrow \mathbf{float} \{B.type = \mathit{float}; B.width = 8; \}$

$C \rightarrow \epsilon \{C.type = t; C.width = w; \}$

$C \rightarrow [\mathit{num}]C_1 \{C.type = \mathit{array}(\mathbf{num.val}, C_1.type);$
 $C.width = \mathbf{num.val} \times C_1.width; \}$

SDT for Array Type

Production

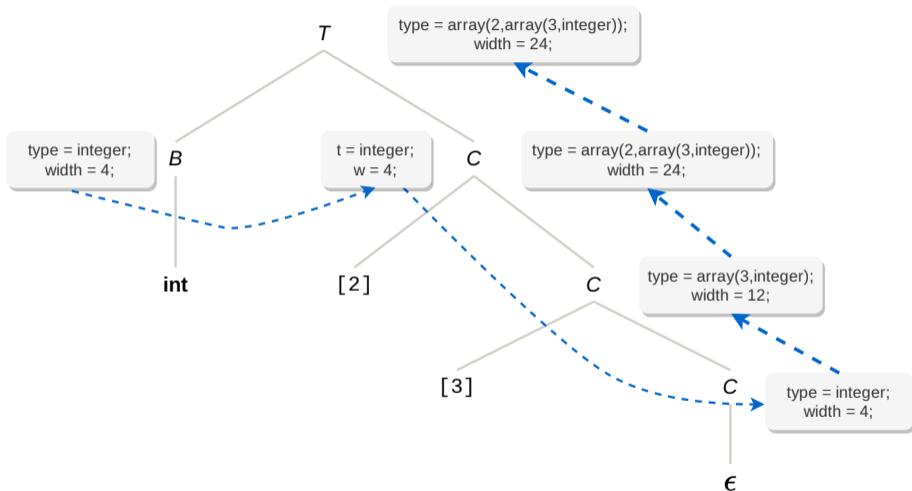
$T \rightarrow BC$

$B \rightarrow \text{int}$




$B \rightarrow \text{float}$

$C \rightarrow \epsilon$

$C \rightarrow [\text{num}]C_1$



References

-  M. Scott. Programming Language Pragmatics. Chapters 7–8, 4th edition, Morgan Kaufmann.
-  A. Aho et al. Compilers: Principles, Techniques, and Tools. Sections 6.3, 6.5, 2nd edition, Pearson Education.
-  K. Cooper and L. Torczon. Engineering a Compiler. Section 4.2, 2nd edition, Morgan Kaufmann.