CS 335: Type Systems

Swarnendu Biswas

Semester 2022-2023-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Type Error in Python

```
def add(x):
      return x + 1

class A(object):
    pass

a = A()
add(a)
```

```
Traceback (most recent call last):
  File "type-error.py", line 8, in <module>
    add1(a)
  File "type-error.py", line 2, in add1
    return x + 1
TypeError: unsupported operand type(s) for +:
'A' and 'int'
```

It is great that compilers help detect such errors

[What is Gradual Typing]

What is a Type?

Set of values and operations allowed on those values

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

Few types are predefined by the programming language

Additional types can be defined by a programmer

- E.g., declaration of a structure in C

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.

Each expression has a type associated with it

The Meaning of Type

Denotational	• A type is a set of values • A value has a given type if it belongs to the set
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
Abstraction-based	• A type is an interface consisting of a set of operations with well-defined and mutually consistent semantics

Type System

- The type of a language construct is denoted by a type expression (e.g., basic types like `int` and `float`)
- 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 equivalence, type compatibility, and type inference
  - Goal is to reduce sources of bugs due to type errors
- Different type systems may be used by different compilers for the same language
  - Pascal includes the index set of arrays in the type information of a function
  - 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 is an error that 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
  - Errors like arithmetic overflow (is a run time error, not a type clash) is outside the scope of type systems
- A type checker implements a type system

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 the 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

Catching Type Errors

- Can a type checker predict that a type error will occur when a particular program is run?
  - Impossible to build a type checker that can predict which programs will result in type errors
- Type checkers make a **conservative approximation** of what will happen during execution
  - Raises error for anything that might cause a type error

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

```
TypeError.java:9: error: incompatible
types: A cannot be converted to int
                add(a);
                    ^
```

[What is Gradual Typing]

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, BCPL)

Manifest Typing
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

Python >>>

```
>>> if False:
...     1 + "two"  # This line never runs, so no TypeError is raised
... else:
...     1 + 2
...
3

>>> 1 + "two"  # Now this is type checked, and a TypeError is raised
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Manifest Typing
What is the difference between statically typed and dynamically typed languages?

Static vs Dynamic Typing

Static	Dynamic
• 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	• Allows fast compilation • Type of a variable can depend on runtime information • Can load new code dynamically • Allows constructs that static checkers would reject

What is Gradual Typing

Categories of Type Systems

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

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

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

```
let val s = “Test”
    val x = 0.0
    val y = 0
…
```

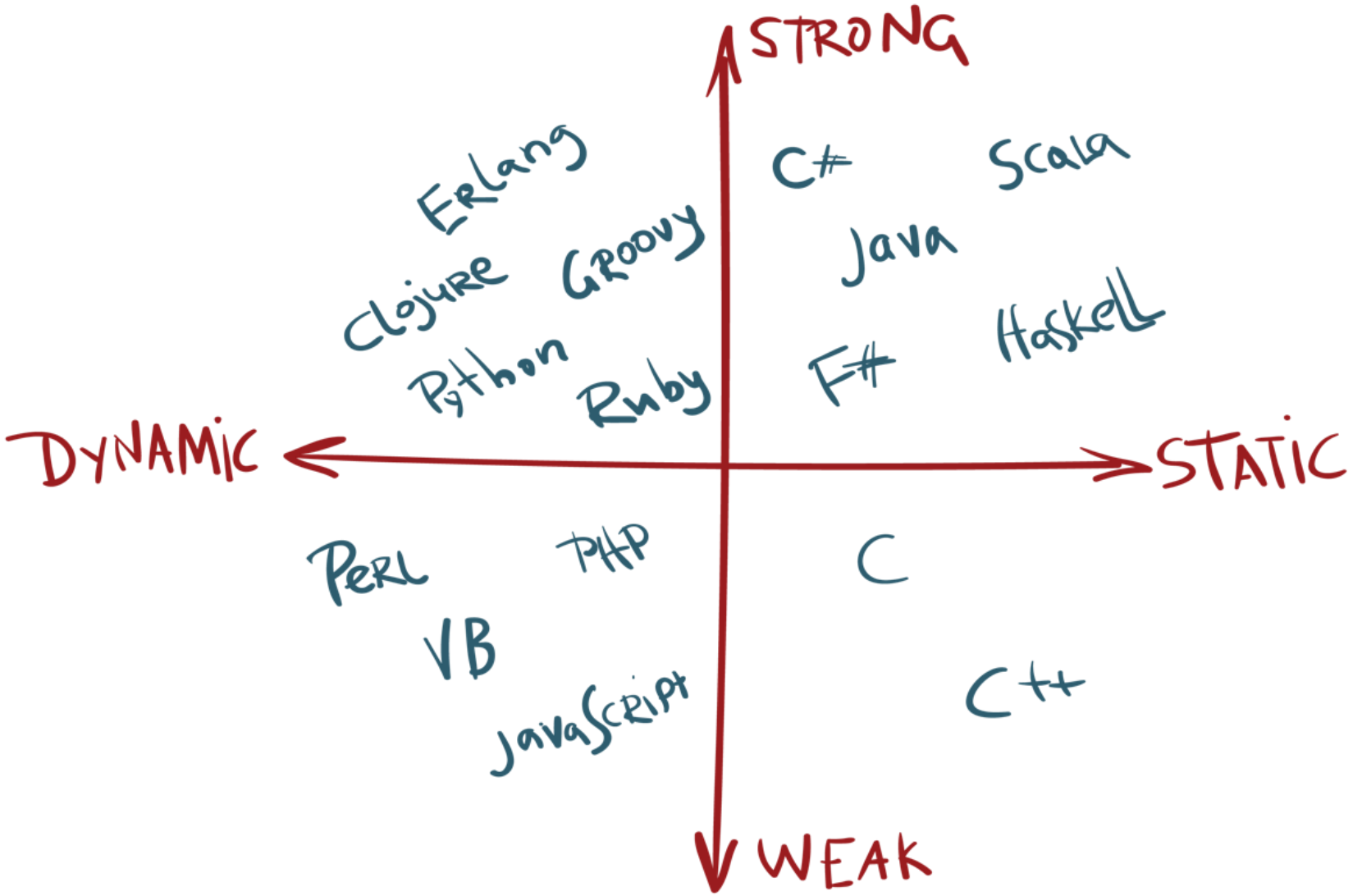
Manifest Typing
Type systems: nominal vs. structural, explicit vs. implicit

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?



[Magic lies here - Statically vs Dynamically Typed Languages]
[Comparison of programming languages by type system]

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 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

[What is Gradual Typing]
[Gradual typing]

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 object is exactly of the given type (**by name**) or is a subtype of that type (e.g., C++, Java, and Swift)

```
class Foo {
  method(input: string): number { ... }
}

class Bar {
  method(input: string): number { ... }
}

let foo: Foo = new Bar(); // ERROR!!
```

- **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(); // Okay.
```

Type Systems: Structural vs. Nominal typing explained
Duck Typing vs Structural Typing vs Nominal Typing

Nominal and Structural Typing

Nominal

```
function greet(person) {
    if (!(person instanceof Person))
        throw TypeError

    alert("Hello, " + person.Name);
}
```

Structural

```
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

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

- 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()
```

Duck Typing vs Structural Typing vs Nominal Typing
Duck typing

TypeScript Example

```
interface Person {
    Name : string;
    Age : number;
}

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
  - Makes use of dynamic duck typing

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

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

Benefits with Types

Usefulness of Types

- Type systems help specify precise program behavior
  - Hardware does not distinguish 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 Runtime Safety

- Well-designed type system helps the compiler detect and avoid run-time errors by identifying misinterpretations of data values
- Type inference – compiler infers a type for each name and expression

Result Types for Addition in Fortran 77				
	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	illegal
complex	complex	complex	illegal	complex

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, compiler needs to maintain type metadata along with value

Implementing Addition

Strongly Typed System

- `integer ← integer + integer`

$$\texttt{IADD}\ R_a, R_b \Rightarrow R_{a+b}$$

- `double ← integer + double`

$$\texttt{I2D}\ R_a \Rightarrow R_{ad}$$
$$\texttt{DADD}\ R_{ad}, R_b \Rightarrow R_{ad+b}$$

Weakly Typed System

```
if type_md(a) == integer
  if type_md(b) == integer
    value(c) = value(a) + value(b)
    type_md(c) = integer
  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
  - 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

- 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)

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 ∪ float ∪ 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

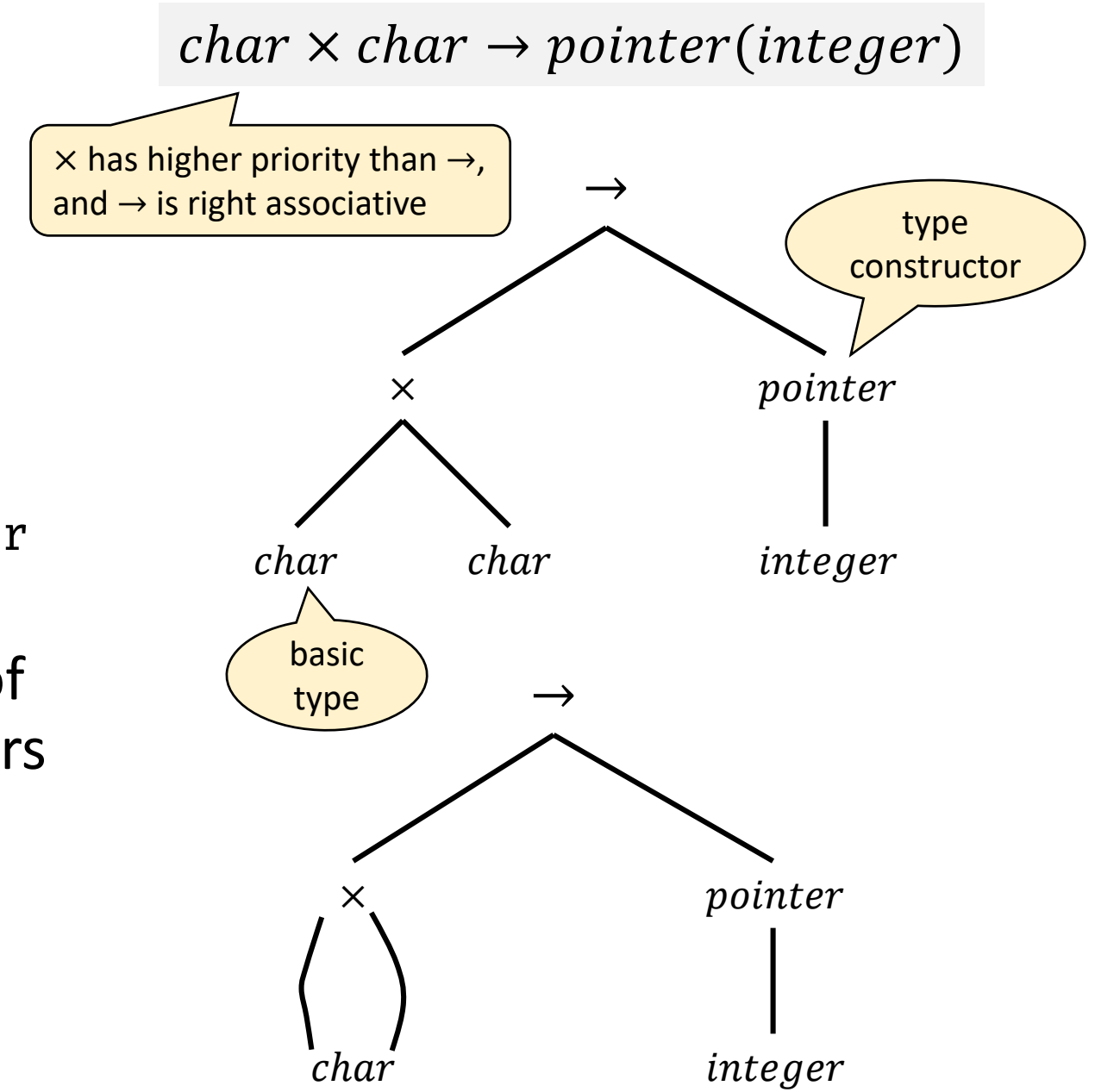
```
int A[10];
```

- 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 signature is a specification of the types of the formal parameters and return value(s) of a function
  - `filter`: $(\alpha \rightarrow \textbf{boolean}) \times$ list of $\alpha \rightarrow$ list of α



Pointer Types

- 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 constructs new pointers

Other Classifications

- Scalar and compound types
  - Scalar indicates a single value, also called simple types
  - Example of compound types are arrays, maps, sets, and structs
  - String in Perl is scalar while it is a compound type in C
- 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 and Coercion Polymorphism

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

```
String fruits = “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 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 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, 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, or
    ii. Are formed by applying same type constructor to equivalent types
    - E.g., OCaml and Haskell

Type Equivalence

Nominal

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

Structural

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

Type Systems: Structural vs. Nominal typing explained

Type Equivalence

Nominal

- Equivalent if same name
  - Identical names can be intentional
  - Can avoid unintentional clashes
  - Difficult to scale for large projects

Structural

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

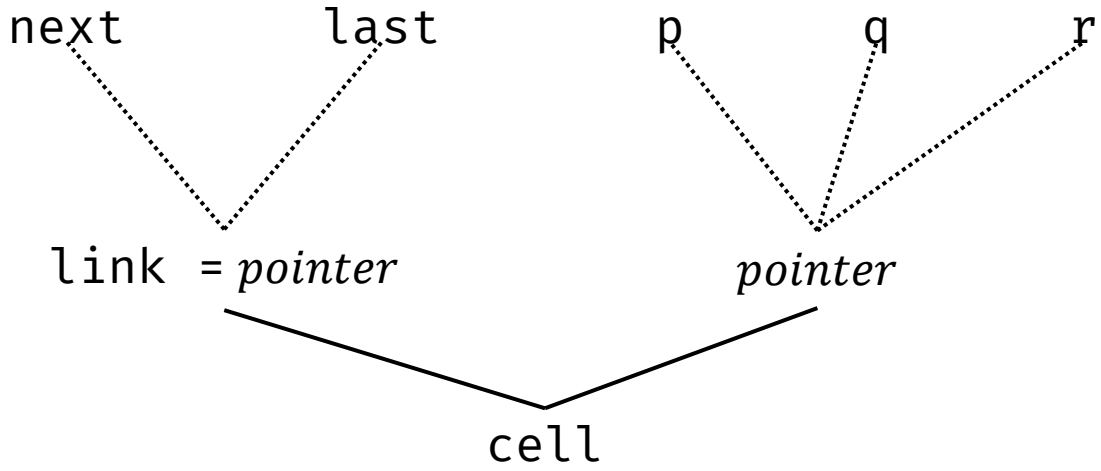
Compilers build trees to represent types

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

Type 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 same type
- Under structural equivalence, all the variables are of 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

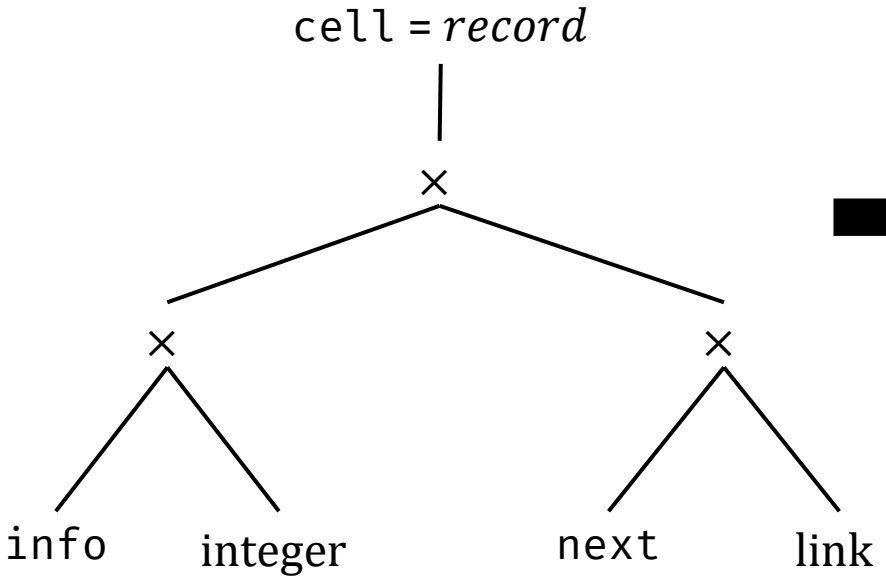
Testing for Structural Equivalence

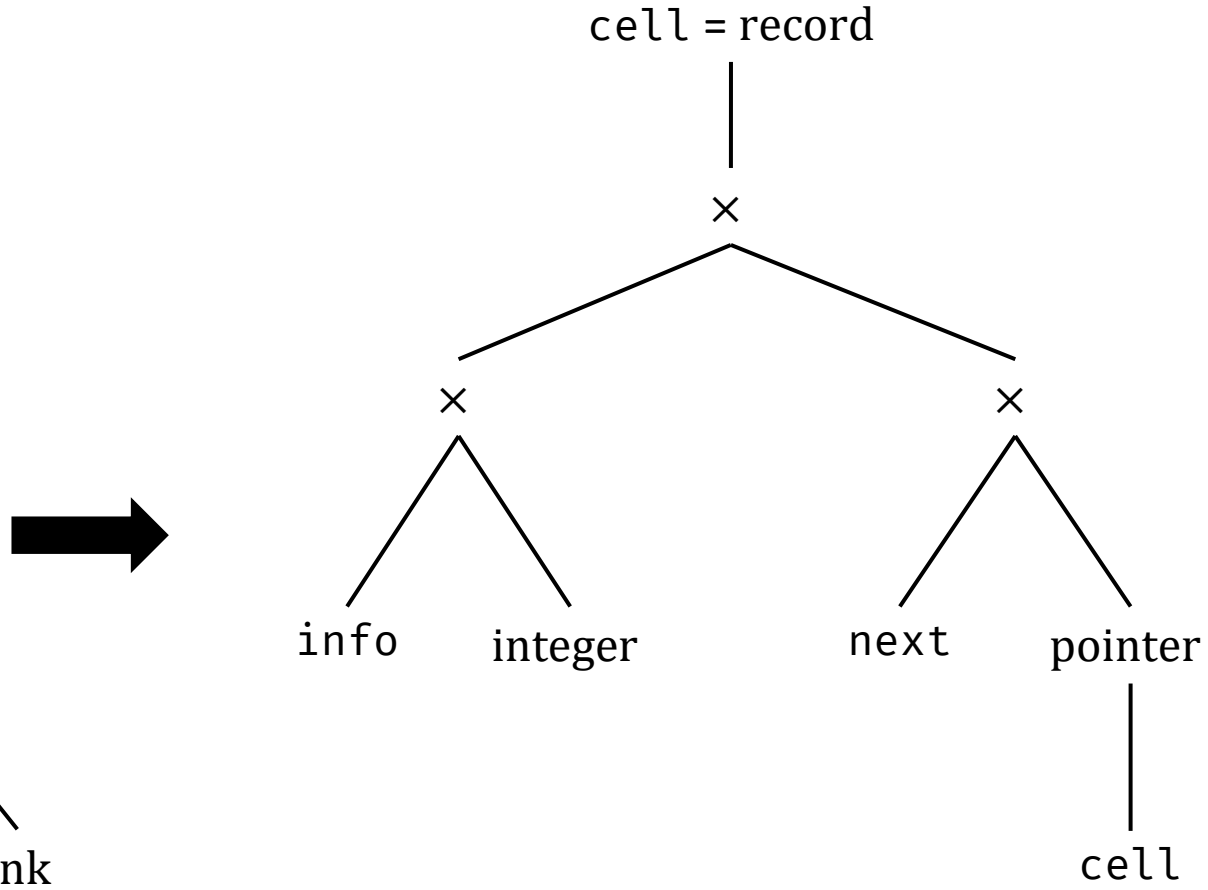
```
boolean struc_equiv(type s, type t)
  if s and t are the same basic type then
    return true
  else if s = array(s1, s2) and t = array(t1, t2) then
    return struc_equiv(s1, t1) and struc_equiv(s2, t2)
  else if s = s1 × s2 and t = t1 × t2 then
    return struc_equiv(s1, t1) and struc_equiv(s2, 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
```

can also ignore array bounds

Representing Recursively-defined Types

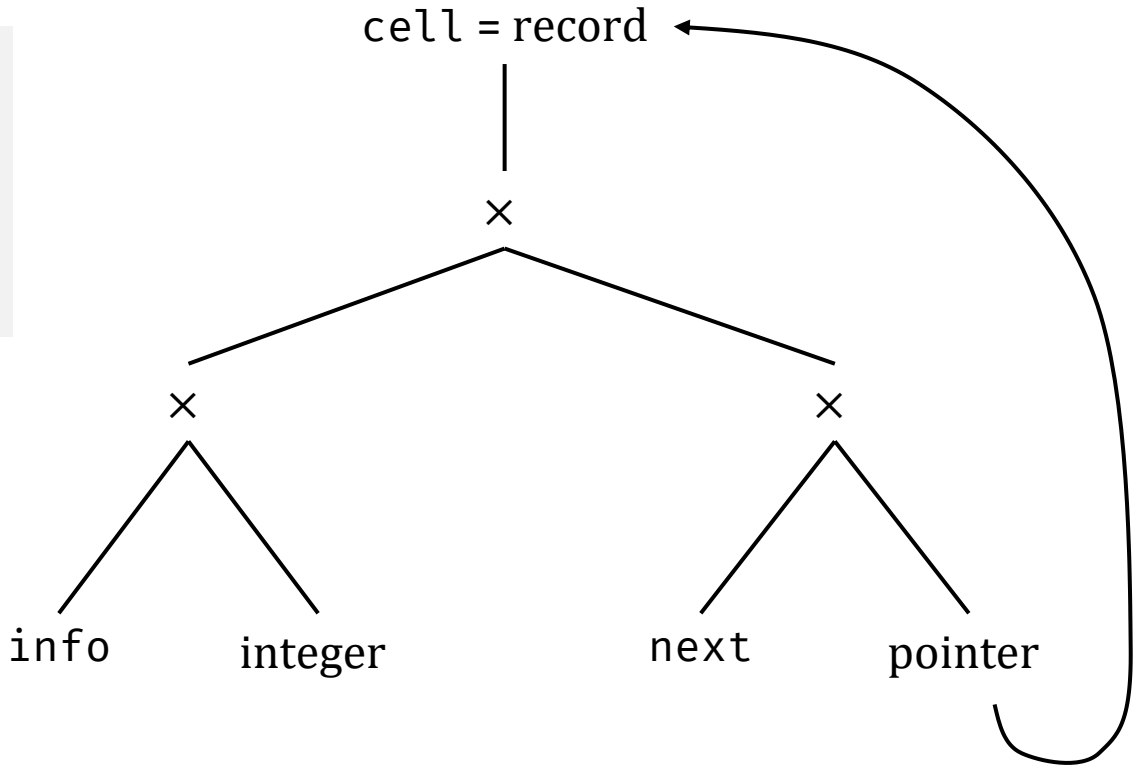
```
type link = ↑ cell;
     cell = record
          info : integer;
          next : link
     end;
```





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
$pointer$	01
$array$	10
$func$	11

Basic Type	Encoding
boolean	0000
char	0001
integer	0010
real	0011

Efficient Encoding of Type Expressions

Type Constructor	Encoding
$pointer$	01
$array$	10
$func$	11

Basic Type	Encoding
boolean	0000
char	0001
integer	0010
real	0011

Type Expression	Encoding
char	000000 0001
$func($**char**$)$	000011 0001
$pointer(func($**char**$))$	000111 0001
$array(pointer(func($**char**$)))$	100111 0001

Six bits used because there are 3 type constructors

Efficient Encoding of Type Expressions

Type Constructor	Encoding
$pointer$	01
$array$	10
$func$	11

Basic Type	Encoding
boolean	0000
char	0001
integer	0010

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

char	000000 0001
$func($**char**$)$	000011 0001
$pointer(func($**char**$))$	000111 0001
$array(pointer(func($**char**$)))$	100111 0001

Type Compatibility

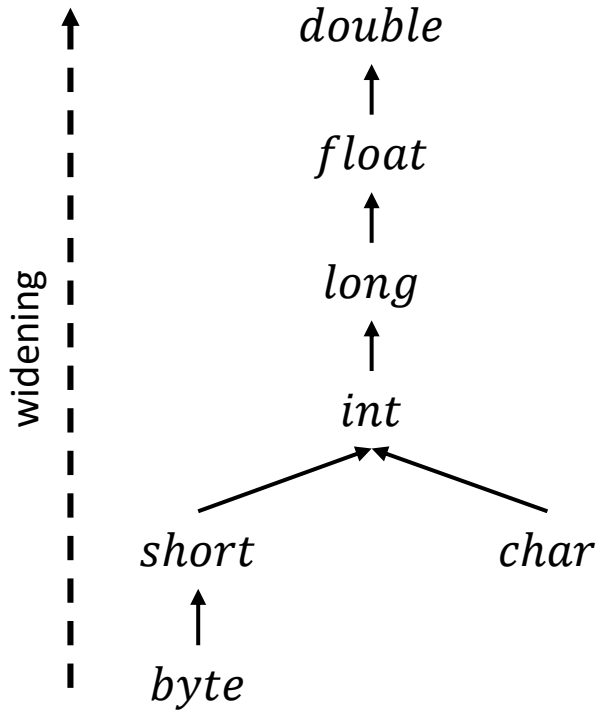
- Most languages do not require equivalence of types in every context
- Type compatibility determines when an object of a certain type can be used in a certain context
- Definitions vary greatly from language to language

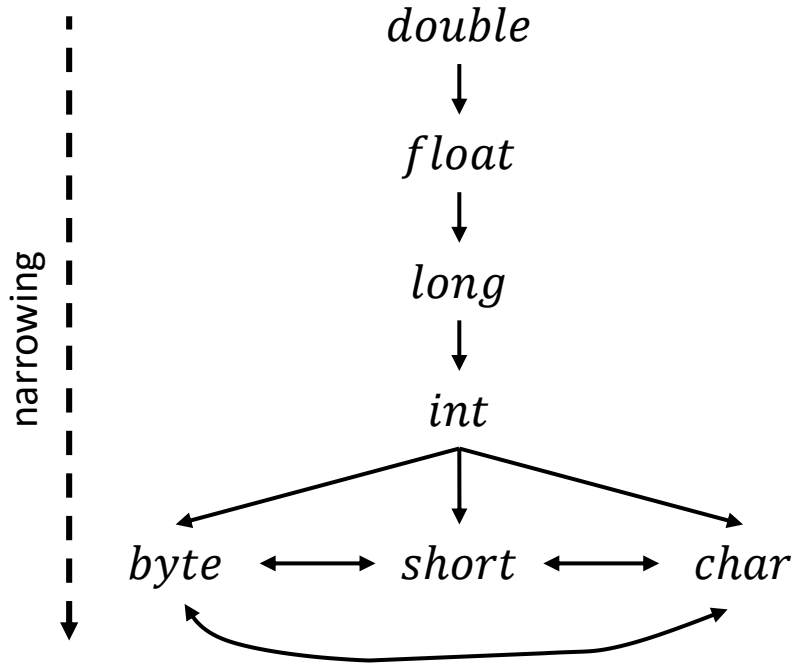
Type Inference Rules

- Specifies, for each operator, the mapping between the operand types and the result type
  - Type of the LHS of an assignment must be same as the RHS
  - In Java, 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 $E_2.type == integer$) $E.type = float$ …}

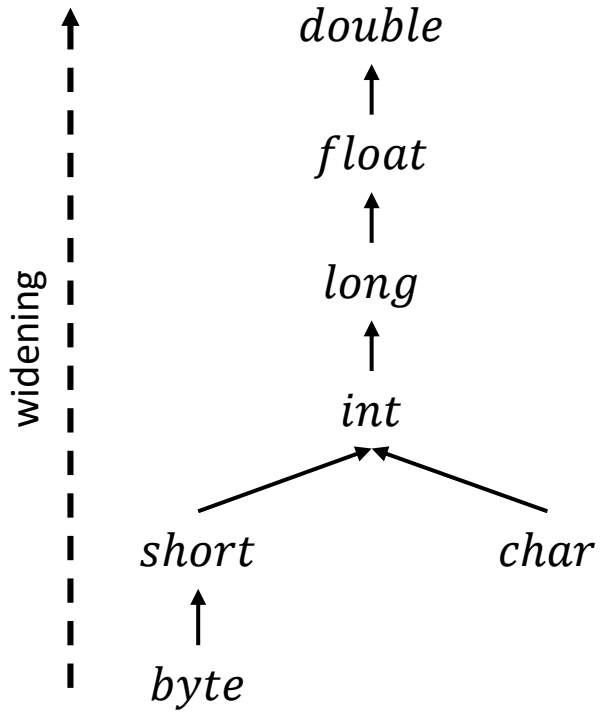




according to precision rules in Java

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 $E_2.type == integer$) $E.type = float$ …}



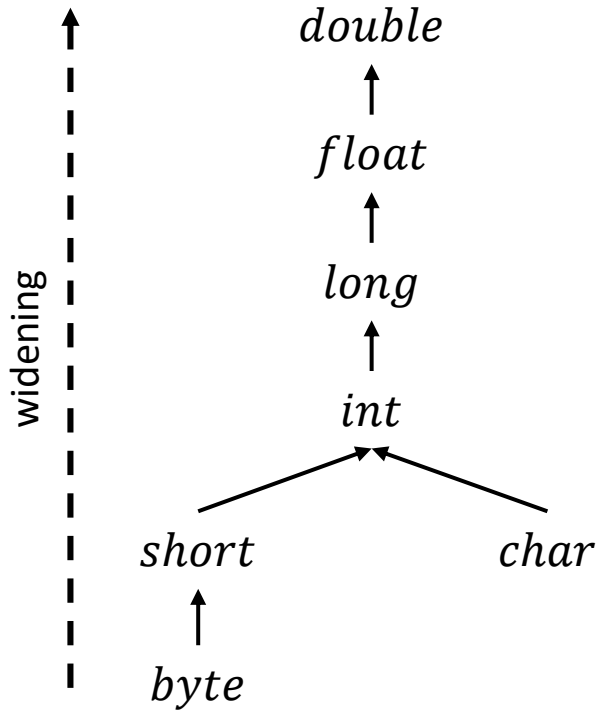
Assume two helper functions:

- $\max(t_1, t_2)$ – return the maximum (or least common ancestor) of the two types in the hierarchy
- $\text{widen}(a, t, w)$ – 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
  error;
}
```

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

Type Synthesis for Overloaded Functions

- Suppose f is a 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 $1 \leq k \leq n$

  ➡ Expression $f(x)$ has type t_k

Type Checking

Type Checking Rules for Coercion from Integer to Real

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

Implicit conversion of constants at compile time can reduce run time

Type Checking

- Some languages allow different levels of checking to apply to different regions of code
- The `use strict` directive in Javascript and Perl applies stronger checking

- Type checking has also been used for checking for system security

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

Specification of a Simple Type Checker

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

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

```
key: integer;
key mod 1999
```

SDT for Manipulating Symbol Table

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

Type Checking of Expressions

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

Type Checking of Statements

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

Production	Semantic Action
$S \rightarrow \textbf{id} = E$	$\{$ **if** id$.type = E.type$ **then** $S.type = void$ **else** $S.type = type\_error$ $\}$
$S \rightarrow$ **if** E **then** S_1	$\{$ **if** $E.type = boolean$ **then** $S.type = S_1.type$ **else** $S.type = type\_error$ $\}$
$S \rightarrow$ **while** E **do** S_1	$\{$ **if** $E.type = boolean$ **then** $S.type = S_1.type$ **else** $S.type = type\_error$ $\}$
$S \rightarrow S_1; S_2$	$\{$ **if** $S_1.type = void$ **and** $S_2.type = void$ **then** $S.type = void$ **else** $S.type = type\_error$ $\}$

Type Checking of Functions

```
int f(double x, char y)
```

$\texttt{f}: double \times char \rightarrow int$

Production	Semantic Action
$E \rightarrow E_1(E_2)$	$\{$ **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
$T \rightarrow BC$
$B \rightarrow$ **int**
$B \rightarrow$ **float**
$C \rightarrow \epsilon$
$C \rightarrow$ [num] C_1

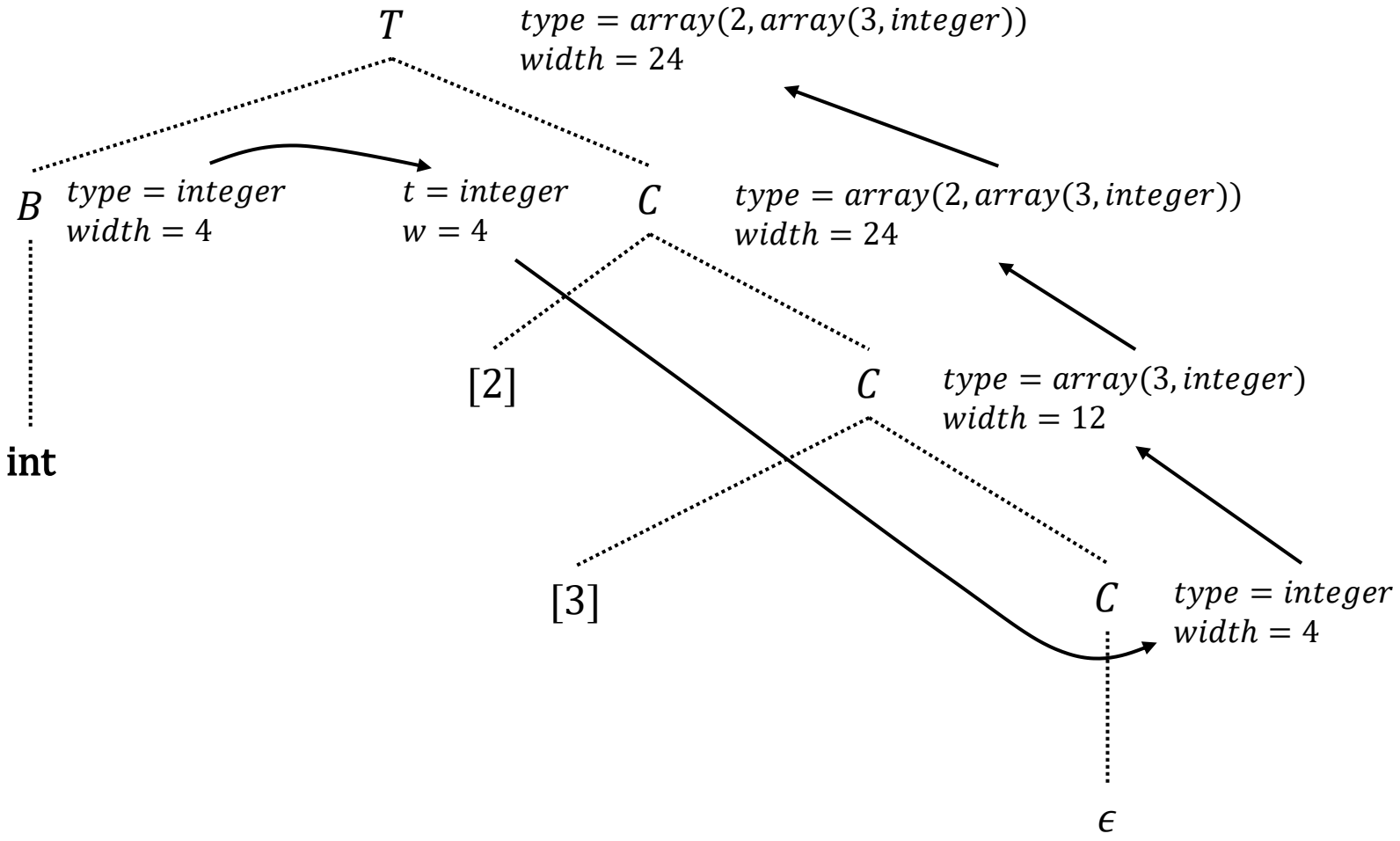
Determine amount of allocation in a declaration

Computing Types and Their Widths

Production	Semantic Action
$T \rightarrow BC$	$T \rightarrow B\ \{ t = B.type; w = B.width; \}$ $C\ \{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow$ **int**	$B \rightarrow$ **int** $\{ B.type = integer;\ B.width = 4;\ \}$
$B \rightarrow$ **float**	$B \rightarrow$ **float** $\{ B.type = float;\ B.width = 8;\ \}$
$C \rightarrow \epsilon$	$C \rightarrow \epsilon\ \{ C.type = t; C.width = w; \}$
$C \rightarrow$ [num] C_1	$C \rightarrow$ [num] $C_1\ \{ C.type = array(\text{num}.val, C_1.type);$ $C.width = num.val \times C_1.width;\ \}$

SDT for Array Type

Production
$T \rightarrow BC$
$B \rightarrow$ **int**
$B \rightarrow$ **float**
$C \rightarrow \epsilon$
$C \rightarrow [\ \text{num}\ ]\ C_1$



References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 1st edition, Chapter 6.
- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 6.2.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 4.2.
- M. Scott. Programming Language Pragmatics, 4th edition, Chapters 7-8.
- Type system