

# Introduction to Type Systems

CSE 340 FALL 2021

Rida Bazzi

RECAP. So far, we have covered the following:

## Syntax

### Lexical Analysis

- getToken() function

### Parsing

- Specifying and identifying the structure of the program
- Context-free grammars: derivations, parse trees, ambiguity
- Recursive-descent parsing
  - inefficiency of backtracking
  - predictive parsing
  - FIRST and FOLLOW sets to drive parsing
  - conditions for the existence of a predictive parser
  - we covered what a general predictive parser looks like in terms of FIRST and FOLLOW sets

## Lambda Calculus

- Studied syntax and semantics of a simple programming language which is the foundation for modern functional programming languages
  - scopes and binding (name resolution)
  - higher-order functions
  - execution semantics (beta reduction, renaming)
  - eager and lazy evaluation (call by value and normal order)
  - recursion
  - How to program with lambda calculus

Turing completeness and the motivation for lambda calculus and Turing machines

## **Basic Semantics**

### **Declarations and scopes**

- declarations
- scoping: static and dynamic
- Resolving references
- Pointer semantics

# TYPE SYSTEMS

Type these set of values and operations that can be applied to value

type compatibility rules specify

- under which conditions an **assignment** is valid
- automatic type casting to be done (if needed)

$$\begin{array}{cc} a & = & b \\ T_1 & & T_2 \end{array}$$

under which constraints on  $T_1$  and  $T_2$  is the assignment valid  
if  $T_1 \neq T_2$ , how do we obtain value of  $a$ ?

type inference rules: what is the type of an "expression" given the types of its constituent parts?

**example:** in the C language,  
if  $x$  has type  $T^*$ ,  $*x$  has type  $T$   
if  $x$  has type  $T$ ,  $\&x$  has type  $T^*$   
 $*(\&x)$  has type  $T$   
if  $x$  is an array of  $T$ ,  $x[i]$  has type  $T$

type declarations

- built-in types
- programmer-defined types

TYPE SYSTEM consists of

- type declaration and
- type compatibility rules and
- type inference rules

# TYPE DECLARATIONS

Programming languages have “constructs” to declare programmer-defined types.

Examples of types that are supported by programming languages include the following (the specifics and the names can vary from one languages to another):

1. Array
2. Structure
3. Class
4. Function
5. Pointer
6. Enumeration
7. Slice (Rust and Golang)

## Named Types

Programming Languages typically allow programmers to give a “name” (ID, identifier) to a declared type.

## Examples

- Class in C++:

```
class C { ... }           // class named C
```

- typedef in C and C++

```
typedef uint8_t BYTE;    // unsigned 8 bit int, BYTE  
                        // is the name
```

- type in Rust

```
type Height = i64;      // signed 64-bit int
```

In all these examples, the programmers gives the type an explicit name. The way the programming language treats these names differ from one language to another.

# TYPE DECLARATIONS

## Anonymous Types

If a type is not explicitly named, we say that it is “anonymous”

this definition is not universal and some languages give specific meaning to anonymous types.

We also say that the type is unnamed (also that is not a universal definition).

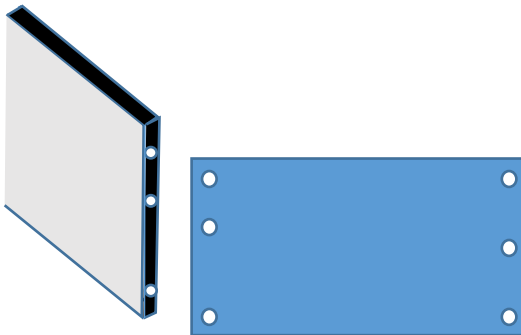
# What are types good for?

- To tame complexity: Abstraction
- Documentation
- Allows for more efficient code generation
- To catch errors at compile-time
- Safety

without strong type checking, you can end up with something like this\*:



The following cannot go wrong



\* Thanks to Jennifer (Spring 2017) for pointing out pictures such as the one above

# TYPE COMPATIBILITY

- Name equivalence
- Internal name equivalence
- Structural equivalence



# Name equivalence

TYPE

```
length : real;  
speed  : real;
```

VAR

```
x : length;           // x and y are not name  
y : speed;           // equivalent because their  
                     // types do not have the same  
                     // names
```

```
{  
    x = 1.0;          allowed constant initialization from  
                     underlying type  
  
    y = x;            not allowed under name equivalence  
}
```

explicitly declared type name

underlying type

An assignment is valid under name equivalence if

- The types of LHS and the RHS of the assignment are the same named type or the same built-in (basic) type.
- The Type of the LHS is a named type and the RHS is a constant value from the underlying type.

# Name equivalence: Examples


TYPE


```
T1 : real;  
T2 : real;  
T3 : array [1..5] of T1;
```


VAR


```
x : array [1..5] of T1;  
y : array [1..5] of T1;  
z, w, : array [1..5] of T1;  
p : T3;  
q : T3;
```


{


```
x = y;  // x and y do not have the same  
// type name. In fact, they do not  
// have a type name!
```

```
z = w;  // z and w do not have the same  
// type name. In fact, they do not  
// have an explicitly declared  
// type name!
```

```
p = q;  // p and q have the same type  
// name
```

```
x[1] = y[1];  // x[1] and y[1] have the same  
// type name which is T1
```

```
p = z;  // p has type named T3  
// z had type with no name  
// they are not name equivalent
```

```
p = z[1];  // p has type named T3  
// z[1] has type named T1
```

}

# Internal Name equivalence

In a programming language, we have seen that names are introduced explicitly or implicitly. Many languages have “declarations” to introduce names. In general, there would be a non-terminal corresponding to a declaration and multiple declarations can be interspersed with statements or appear in declaration lists.

For example, in the C language

```
int i;  
int j,k;
```

are two separate declarations. One declaration introduces i and one declaration introduces j and k. j and k are part of the same declaration, but i and j are not part of the same declaration.

When a type does not have a name, the compiler provides an internal name that is not visible to the program, but that is used by the compiler for type checking. Names that appear in the same declaration will get the same internal type name.

For example,

```
struct {int a; int b;} x, y;
```

introduces two variables x and y that have the same internal name.

Variables that have the same programmer defined type name will also have the same internal name. This is the motivation for the following definition.

**Definition:** x and y are internal-name equivalent if

- They are name equivalent or
- They are part of the same declaration (same internal name)

under no other conditions are they internal name equivalent (note that this definition is not standard)

Example

TYPE

```
T1 : int;  
T2 : int;
```

VAR

```
s : array [1 ... 4] of int;  
t : array [1 ... 4] of int;  
u, v : array [1 ... 4] of int;  
w : int;  
x : T1;  
y : T1;  
z : T2;
```

u and v are internal name equivalent (same declaration)

x and y are internal name equivalent (same type name)

no other variables declared above are internal name equivalent

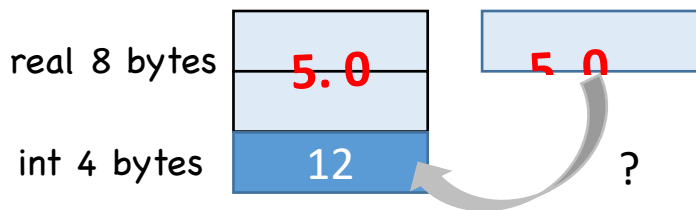
in particular s and t are not internal name equivalent because they do not have a type name and they are not part of the same declaration.

# Structural Equivalence

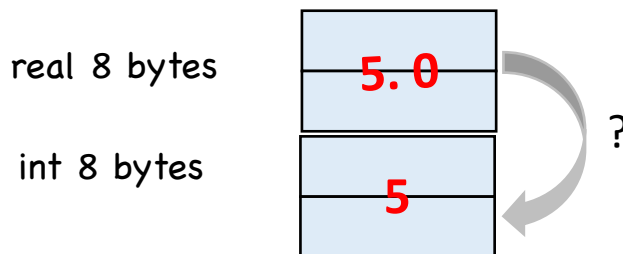
## Informal motivation

informally, structural equivalence allows an assignment if the assignment can be made with memory copy without “breaking” the representation of data types directly or indirectly

int and real **are not** structurally equivalent because they are different basic types. Copying an int value to a location that is supposed to store a real value or vice versa will result in breaking the data representation: what does copying the first 4 bytes from the real value to the location that stores an int value mean? How about copying the int value to a location that stores real values?



The issue is not just that the sizes are different. The representations are also different.



The two values have completely different representations. Copying 8 bytes that represent a real value to a location that is supposed to hold an integer value and then reading the resulting value as int will give a value that is unrelated to 5.

# Structural Equivalence

## Informal motivation

informally, structural equivalence allows an assignment if the assignment can be made with memory copy without “breaking” the representation of data types directly or indirectly

## Example

### TYPE

```
T1 : int;  
T2 : real;  
T3 : int;
```

### VAR

```
a : T1;  
b : T2;  
c : T3;  
  
p : pointer to T1;  
q : pointer to T2;
```

a & c **are** structurally equivalent because both are equivalent to int

a & b **are not** structurally equivalent because one is equivalent to int and the other is equivalent to real.

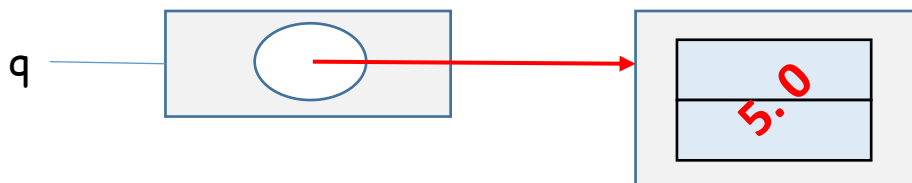
p & q **are not** structurally equivalent because the types “they point to” are not structurally equivalent

# Pointers Structural Equivalence

Pointers are structurally equivalent if the types they point to are structurally equivalent. To understand why this is the case, consider  $p$  and  $q$  from the previous page

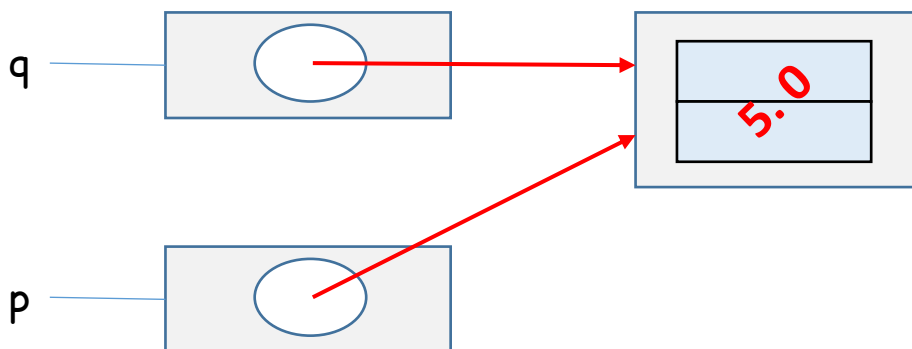
if we execute  $*q = 5.0;$

we get the following picture (assuming memory is previously allocated)



What happens if we allow  $p = q;$

We get the following picture



Now, the value of  $p$  is the address of a location containing a real value

if we execute  $a = *p;$

we will try to read an integer value from a location that holds a real value which breaks the representation because  $p$  will be interpreted as the address of a location that contains an integer.

# Structural Equivalence: Rules

T1 and T2 are structurally equivalent if and only if one of the following rules is satisfied

0) T1 is declared to be T2 in a type declaration.

## Example

```
TYPE  
    T1 : T2;
```

1) T1 and T2 are structurally equivalent to the same basic type.

2) T1 : pointer to t1  
 T2 : pointer to t2;

and t1 and t2 are structurally equivalent.

# Structural Equivalence: Rules

3) T1 : struct {  
     $a_1 : t_1;$   
     $a_2 : t_2;$   
    .  
    .  
     $a_k : t_k;$  }

T2 : struct {  
     $a'_1 : t'_1;$   
     $a'_2 : t'_2;$   
    .  
    .  
     $a'_k : t'_k;$  }

and  $t_i$  &  $t'_i$  are structurally equivalent for  $i = 1 \dots k$



# Structural equivalence: remark

I should note here that this definition (for structures) requires that the two types have the same number of fields because  $k$  is the same for both. Also, it requires that the types of the field in the order in which they appear are structurally equivalent to the corresponding fields in the other structure.

for example

```
T1 : struct {  
    a : int;  
    b : real;  
}
```

```
T2 : struct {  
    b : real;  
    a : int;  
}
```

cannot be shown to be structurally equivalent according to 3) because the types in the order they appear are not structurally equivalent to the corresponding types (same order) in T2

$t_1 = \text{int}$	$t'_1 = \text{real}$	are not structurally equivalent
$t_2 = \text{real}$	$t'_2 = \text{int}$	are not structurally equivalent


It is enough for one  $t_i$  to be not structurally equivalent to the corresponding  $t'_i$  for the types to be not structurally equivalent

Note that since T1 and T2 are structures, this is the only rule that can apply and if it does not apply, they are not structurally equivalent.

If T1 is a structure and T2 is a pointer, they are not structurally equivalent because there is no rule that allows a pointer and a structure to be structurally equivalent

# structural equivalence **rules**: arrays

- 4)  $T1 : \text{array } \text{range}_1 \times \text{range}_2 \times \dots \times \text{range}_d \text{ of } t$   
 $T2 : \text{array } \text{range}'_1 \times \text{range}'_2 \times \dots \times \text{range}'_d \text{ of } t'$

 cartesian product

Here T1 and T2 are multi-dimensional arrays. Ranges can be specified as

- [ low , high ] : The number of entries in the dimension for the range is  $hi - low + 1$ .
- [n] : The number of entries in the dimension for the range is n (the range being from 0 to n-1 or 1 to n).

T1 and T2 are structurally equivalent if **all** of the following holds

same number of dimension (this is captured by having the same d for both)

element type are structurally equivalent:  $t$  structurally equivalent to  $t'$

the  $i$ 'th dimension in T1 and the  $i$ 'th dimension in T2 have the same number of entries:  $\text{range}_i$  and  $\text{range}'_i$  have the same number of entries (see above)

## example

$T1 : \text{array } [1..4][1..5] \text{ of int}$   
 $T2 : \text{array } [1..5][1..4] \text{ of int}$   
 $T3 : \text{array } [1..4][1..5] \text{ of real}$   
 $T4 : \text{array } [2..5][2..6] \text{ of int}$

T1 & T4 are structurally equivalent

T2 is not structurally equivalent to any of the other types

T3 is not structurally equivalent to any of the other types

Note that arrays cannot be structurally equivalent to other types (pointers, structures, functions, ...)

# Function Types

Consider the following three declarations (in C):

```
int x;
```

```
int f(int);
```

```
int g(float);
```

The type of x is clearly int.

The type of f is not int. int is the return type of f. The type of f is: function from int to int or **int -> int**

The type of g is not int. int is the return type of g. The type of g is: function from float to int or **float -> int**

# structural equivalence **rules**: functions

- 5)  $T1$  : function of  $(t_1, t_2, \dots, t_m)$  returns  $t$   
 $T2$  : function of  $(t'_1, t'_2, \dots, t'_m)$  returns  $t'$

$T1$  and  $T2$  are structurally equivalent if all of the following hold

parameters in the order they appear are structurally equivalent:  $t_i$  &  $t'_i$  are structurally equivalent

return types are structurally equivalent:  $t$  &  $t'$  are structurally equivalent

## examples

- $T1$  : function of (int, int) returns int  
 $T2$  : function of (int, real) returns int  
 $T3$  : function of (int, int) returns real  
 $T4$  : function of (real, int) returns int  
 $T5$  : function of (int, int ) returns pointer to int  
 $T6$  : function of (int, int) returns int

Only  $T1$  and  $T6$  are structurally equivalent.

Note that functions cannot be structurally equivalent to other types (arrays, structures, pointers, ....)

# How do we check for structural equivalence?

## Initialization

We start by assuming all types are structurally equivalent.

We mark obviously non-equivalent types as non-equivalent. For example:

- function types not structurally equivalent to array types,
- structure types are not structurally equivalent to basic types
- structure types with different number of elements are not structurally equivalent

We store this information in a Boolean matrix  $E$  of size  $m \times m$  where  $m$  is the number of types we are considering.

If an entry  $E[i,j]$  is true, this means that, so far, we think type  $i$  and type  $j$  are equivalent. If an entry  $E[i,j]$  is false, this means that we have determined that type  $i$  and type  $j$  are not structurally equivalent

# How do we check for structural equivalence?

## Main Loop

repeat

    change = false

    for every pair of types  $i$  and  $j$  for which  $E[i,j] = \text{true}$  do

        if a rule for structural equivalence is  
        satisfied by type  $i$  and type  $j$   
            // no change

        if none of the rules for equivalence is  
        satisfied for type  $i$  and type  $j$   
             $E[i,j] = \text{false}$   
            change = true

until change = false

# Example 1

```
A: struct {  
    a : int;  
    next : pointer to B;  
}
```

```
B: struct {  
    a : int;  
    next : pointer to C;  
}
```

```
C : struct {  
    a : real;  
    next : pointer to A;  
}
```

```
T1 : array [1..4] of A;  
T2 : array [1..4] of B;
```

Initially we assume that all three structures are equivalent and all arrays are equivalent, but structures and arrays are not equivalent (effectively we have two tables one for structures and one for arrays)

	A	B	C	T1	T2
A	T	T	T	F	F
B	T	T	T	F	F
C	T	T	T	F	F
T1	F	F	F	T	T
T2	F	F	F	T	T

# Main loop

We compare  $A$  to all other structures

$A \equiv B$  ?  $\text{int} \equiv \text{int}$  (same basic type)  
pointer to  $B \equiv$  pointer to  $C$  (from table)

$A \equiv C$  ?  $\text{int} \not\equiv \text{real}$  (different basic types)

so  $A$  and  $C$  are not equivalent and we update the table

	A	B	C	T1	T2
A	T	T	<b>F</b>	<b>F</b>	<b>F</b>
B	T	T	T	<b>F</b>	<b>F</b>
C	<b>F</b>	T	T	<b>F</b>	<b>F</b>
T1	<b>F</b>	<b>F</b>	<b>F</b>	T	T
T2	<b>F</b>	<b>F</b>	<b>F</b>	T	T



# Main loop

then we compare B to C

$B \equiv C$  ?  $\text{int} \not\equiv \text{real}$  (different basic types)

so B and C are not equivalent and we update the table

	A	B	C	T1	T2
A	T	T	F	F	F
B	T	T	F	F	F
C	F	F	T	F	F
T1	F	F	F	T	T
T2	F	F	F	T	T

# Main loop

Next, we compare the two arrays

$T1 \equiv T2$  ?

1 = 1 same number of dimensions

4 = 4 same number of entries in each dimension

$A \equiv B$  element types are equivalent (according to table)  
this results in no change

Since there has been changes in the first pass, we do a second pass.

$A \equiv B$  ?  $\text{int} \equiv \text{int}$  (same basic type)  
pointer to B  $\neq$  pointer to C  
because B and C are not equivalent (from table)

	A	B	C	T1	T2
A	T	F	F	F	F
B	F	T	F	F	F
C	F	F	T	F	F
T1	F	F	F	T	T
T2	F	F	F	T	T

# Main loop

Next, we compare the two arrays

$T1 \equiv T2$  ?

1 = 1 same number of dimensions

4 = 4 same number of entries in each dimension

$A \not\equiv B$  element types are not equivalent (according to table)

and we update the table

	A	B	C	T1	T2
A	T	F	F	F	F
B	F	T	F	F	F
C	F	F	T	F	F
T1	F	F	F	T	F
T2	F	F	F	F	T

one more pass results in no more changes.

# Example 2

A = struct { int a; struct B \* next }

B = struct { int a; struct A \* next }

We initialize the table as follows

	A	B
A	T	T
B	T	T

If we compare A and B, we get

int  $\equiv$  int

pointer to B  $\equiv$  pointer to A because according to the table

A  $\equiv$  B

So, after one pass, there is no change and A and B are structurally equivalent