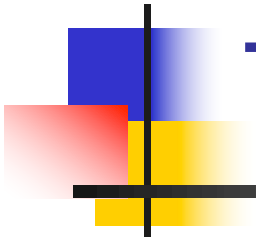


Types

Read: Scott, Chapters 7.1-7.2 and 8

Types and Type Systems

- A key concept in programming languages
- Haskell's type system
- Today, a more pragmatic view of types

Lecture Outline

- *Types*
- *Type systems*
  - *Type checking*
  - *Type safety*
- *Type equivalence*
- *Types in C (next time)*

- *Primitive types (next time)*
- *Composite types (next time)*

What Is a Type?

- A name for a set of related values and the valid operations on those values
  - Integers:

    +, -, *, /, <, <=, ==, ,>=, >
  - Arrays:

    lookUp(<array>,<index>)

    assign(<array>,<index>,<value>)

    initialize(<array>), setBounds(<array>)
  - User-defined types:

    Java interfaces

What Is the Role of Types?

- What is the role of types in programming languages?
  - Semantic correctness 1 + True ✗
  - Data abstraction
    - Abstract Data Types (as in PSoft), Algebraic Data Types (as in Haskell)
  - Documentation (static types only)

Three Views of Types

- Denotational (or set) point of view:
  - A type is simply a set of values. A value has a given type if it belongs to the set. E.g.
    - int = { **…-1,0,1,2,...** }
    - char = { '**a**','**b**',... }
    - bool = { true, false }

- Abstraction-based point of view:
  - A type is an interface consisting of a set of operations with well-defined meaning

Three Views of Types

- Constructive point of view:
  - Primitive/simple types: e.g., int, char, bool
  - Composite/constructed types:
    - Constructed by applying type constructors
    - pointer e.g., **pointerTo**(int)
    - array e.g., **arrayOf**(char) or **arrayOf(**char,20) or ...
    - record/struct e.g., **record**(age:int, name:arrayOf(char))
    - union e.g. **union**(int, pointerTo(char))

    CAN BE NESTED! pointerTo(arrayOf(pointerTo(char**)))**
- For most of us, types are a mixture of these three views

What Is a Type System?

- A mechanism to define types and associate them with programming language constructs
  - Deduce types for program constructs
  - Deduce if a construct is “type correct” or “type incorrect”

- Additional rules for type equivalence, type compatibility
  - Important from pragmatic point of view

What Is a Type System?

$$\frac{}{\Gamma \vdash n : \{x : \mathtt{int} \mid x = n\}}\ (\text{T-Const}) \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\text{T-Var}) \qquad \frac{[\![\Gamma]\!] \models \mathit{false}}{\Gamma \vdash \mathtt{fail} : T}\ (\text{T-Fail})$$

$$\frac{\Gamma, a_i \vdash e_i : T \quad \text{for each } i \in \{1, \ldots, n\}}{\Gamma \vdash \oplus\{a_1 \Rightarrow e_1, \ldots, a_n \Rightarrow e_n\} : T}\ (\text{T-Branch}) \qquad \frac{\Gamma, x : \mathtt{int} \vdash e : T}{\Gamma \vdash \mathtt{let}\ x = *\ \mathtt{in}\ e : T}\ (\text{T-Rand})$$

$$\frac{\Gamma, x : \{y : \mathtt{int} \mid y = a\} \vdash e : T}{\Gamma \vdash \mathtt{let}\ x = a\ \mathtt{in}\ e : [a/x]T}\ (\text{T-AExp}) \qquad \frac{\Gamma \vdash e : T' \qquad \Gamma \vdash_s T' <: T}{\Gamma \vdash e : T}\ (\text{T-Sub})$$

$$\frac{[\![\Gamma]\!], a_1 \models a_2}{\Gamma \vdash_s \{x : \mathtt{int} \mid a_1\} <: \{x : \mathtt{int} \mid a_2\}}\ (\text{S-Int}) \qquad \frac{\Gamma \vdash_s T_{21} <: T}{\Gamma \vdash_s (x : T_1)}$$

$$\frac{\Gamma \vdash y : (z : T_1) \to T_2 \qquad \Gamma \vdash z : T_1 \qquad \Gamma, x : T}{\Gamma \vdash \mathtt{let}\ x = y\ z\ \mathtt{in}\ e : T}$$

$$\frac{\Gamma(\mathtt{main}) = (x : \mathtt{int}) \to \mathtt{int} \qquad x_1 : T_1, \ldots, x_k : T_k \vdash e : T \text{ for each } f : (x_1 : T_1) \to \cdots \to (x_k : T_k) \to T}{\vdash \mathbb{P} : \Gamma}$$

Rules for reading and writing: $(\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \circ\!\!-^{\tau}\ \tau_2$ and $\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau}\ \tau_2)$

$$\frac{\tau\ \mathrm{copy}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n\ \tau \circ\!\!-^{\tau}\ \mathbf{own}_n\ \tau}\ (\text{T-read-own-copy}) \qquad \frac{n = \mathrm{size}(\tau)}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{own}_m\ \tau \circ\!\!-^{\tau}\ \mathbf{own}_m\ \not\downarrow n}\ (\text{T-read-own-move})$$

$$\frac{\tau\ \mathrm{copy} \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa\ \mathrm{alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mu}\ \tau \circ\!\!-^{\tau}\ \&^{\kappa}_{\mu}\ \tau}\ (\text{T-read-bor})$$

$$\frac{\mathrm{size}(\tau) = \mathrm{size}(\tau')}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n\ \tau' \multimap^{\tau}\ \mathbf{own}_n\ \tau}\ (\text{T-write-own}) \qquad \frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa\ \mathrm{alive}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mathbf{mut}}\ \tau \multimap^{\tau}\ \&^{\kappa}_{\mathbf{mut}}\ \tau}\ (\text{T-write-bor})$$

$[\![\emptyset]\!] = \mathit{true}, \quad [\![\Gamma, x : \{y : \mathtt{int} \mid a\}]\!] = [\![\Gamma]\!] \wedge [x/y]a, \quad [\![\Gamma, a]\!] = [\![\Gamma]\!] \wedge$

Rules for typing of instructions: $(\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T} \vdash I \dashv x.\, \mathbf{T}_2)$

Typir

$$\frac{\begin{array}{l}(\forall i \in \{1, \ldots, n\}) \qquad D; A_i \vdash e_i[x_1 := x'_1] \cdots [x_n := x'_n] : v_i \\ (\forall j \in \{j_1, \ldots, j_m\}) \quad \mathrm{Gen}(A_1 + \cdots + A_n, v_j) \leq_{\forall 2,1} ui_j\end{array}}{D, x_1 : \forall \overrightarrow{\alpha^1}.\langle A_0, v_1\rangle, \ldots, x_n : \forall \overrightarrow{\alpha^n}.\langle A_0, v_n\rangle; A' \vdash e[x_1 := x'_1] \cdots [x_n := x'_n] : v}\ (\text{S-num})$$

$$\frac{}{D; A \vdash \mathtt{letrec}\ \{x_1 = e_1, \cdots, x_n = e_n\}\ \mathtt{in}\ e : v}\ (\text{LetrecPsI})$$

where $x'_1, \ldots, x'_n$ are fresh,
$A_1 + \cdots + A_n = A_0,\ x'_{j_1} : ui_{j_1}, \ldots, x'_{j_m} : ui_{j_m},$
$\mathrm{Dom}(A_0) \cap \{x'_1, \ldots, x'_n\} = \emptyset,$
$(\forall i \in \{1, \ldots, n\})\ \{\overrightarrow{\alpha^i}\} = \mathrm{FTV}(A_0) \cup \mathrm{FTV}(v_i),$
$\{x'_{h_1}, \ldots, x'_{h_q}\} = \{x'_1, \ldots, x'_n\} - \mathrm{Dom}(A'),$ and
$A = A' + A_{h_1} + \cdots + A_{h_q}$
$(\forall i \in \{1, \ldots, n\})\quad (\mathbf{Ind}^{\rightarrow}(v_1), \ldots, \mathbf{Ind}^{\rightarrow}(v_n)) =$
$\mathbf{Min}\{\ (\mathbf{Ind}^{\rightarrow}(v'_1), \ldots, \mathbf{Ind}^{\rightarrow}(v'_n)) \mid$
$D; A'_1 \vdash e_1[x_1 := x'_1] \cdots [x_n := x'_n] : v'_1,$
$\vdots$
$D; A'_n \vdash e_n[x_1 := x'_1] \cdots [x_n := x'_n] : v'_n,$
$A'_1 + \cdots + A'_n = A'_0,\ x'_{j_1} : ui'_{j_1}, \ldots, x'_{j_m} : ui'_{j_m},$
$(\forall j \in \{j_1, \ldots, j_m\})\ \mathrm{Gen}(A'_1 + \cdots + A'_n, v'_i) \leq_{\forall 2,1} ui'_i\ \}$

$$\frac{}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p_1 \lhd \mathbf{int}, p_2 \lhd \mathbf{int} \vdash p_1 \leq p_2 \dashv x.\, x \lhd \mathbf{bool}}\ (\text{S-nat-leq})$$

$$\frac{n = \mathrm{size}(\tau)}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid p \lhd \mathbf{own}_n\ \tau \vdash \mathbf{delete}(n, p) \dashv \emptyset}\ (\text{S-delete})$$

$$\frac{\overline{\tau}_i = \tau \qquad \tau_1 \multimap^{\Sigma\overline{\tau}} \tau'_1}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 :=^{\mathrm{inj}\ i} p_2 \dashv p_1 \lhd \tau'_1}\ (\text{S-sum-assgn})$$

rules of λ_{Rust} (helper judgments and instructions).

What Is a Type System?

STATIC

1) RULES (INTERPRETATION) ASSOCIATED TO SYNTACTIC CONSTRUCTS; DECIDE IF CONSTRUCT IS TYPE CORRECT or TYPE INCORRECT

2) DESCRIPTION OF PROGRAM EXECUTION / EVALUATION

→ FORBIDDEN ERRORS

1 + True : FORBIDDEN IN HASKELL

1/0 : EXCEPTION

3) TYPE SAFETY THEOREM THAT LINKS 1) and 2)

"If program is type correct, it cannot go wrong."

"Well typed programs do not go wrong."

What Is Type Checking?

- The process of ensuring that the program obeys the type rules of the language

- Type checking can be done statically
  - At compile-time, i.e., before execution
  - Statically typed (or statically checked) language
- Type checking can be done dynamically
  - At runtime, i.e., during execution
  - Dynamically typed (or dynamically checked) language

What Is Type Checking?

- Statically typed (better term: statically checked) languages
  - Typically require type annotations (e.g., A a, List<A> list)
  - Typically have a complex type system, and most of type checking is performed statically (at compile-time)
    - Ada, Pascal, Java, C++, Haskell, ML/OCaml TypeScript
  - A form of early binding
- Dynamically typed (better term: dynamically checked) languages. Also known as Duck typed…
  - Typically require no type annotations!
  - All type checking is performed dynamically (at runtime)
    - Smalltalk, Lisp and Scheme, Python, JavaScript

What Is Type Checking?

- The process of ensuring that the program obeys the type rules of the language
- Type safety
  - Textbook defines term prohibited application (also known as forbidden error): intuitively, a prohibited application is an application of an operation on values of the wrong type
  - Type safety is the property that no operation ever applies to values of the wrong type at runtime. I.e., no prohibited application (forbidden error) ever occurs

Language Design Choices

- Design choice: what is the set of forbidden errors?
  - Obviously, we cannot forbid all possible semantic errors…
  - Define a set of forbidden errors
- Design choice: Once we've chosen the set of forbidden errors, how does the type system prevent them?
  - Static checks only? Dynamic checks only? A combination of both?
- Furthermore, are we going to absolutely disallow forbidden errors (be type safe), or are we going to allow for programs to circumvent the system and exhibit forbidden errors (i.e., be type unsafe)?

Forbidden Errors

- Example: indexing an array out of bounds
  - **`a[i], a is of size Bound, i<0 or Bound≤i`**
  - In C, C++, this is not a forbidden error
    - **`0≤i`** and **`i<Bound`** is not checked (bounds are not part of type)
    - What are the tradeoffs here? + FLEXIBLE CODE, FAST − UNSAFE
  - In Pascal, this is a forbidden error. Prevented with static checks
    - **`0≤i`** and **`i<Bound`** must be checked at compile time
    - What are the tradeoffs here? + SAFE, FAST − INFLEXIBLE
  - In Java, this is a forbidden error. It is prevented with dynamic checks
    - **`0≤i`** and **`i<Bound`** must be checked at runtime
    - What are the tradeoffs here? + SAFE, FLEXIBLE − SLOW

Type Safety

- Java vs. C++:
  - Java: `Duck q; …; q.quack()` class `Duck` has `quack`
  - C++: `Duck *q; …; q->quack()` class `Duck` has `quack`

  Can we write code that passes the type checker, and yet it calls `quack()` on an object that isn't a `Duck` at runtime?
    - In Java?
    - In C++?

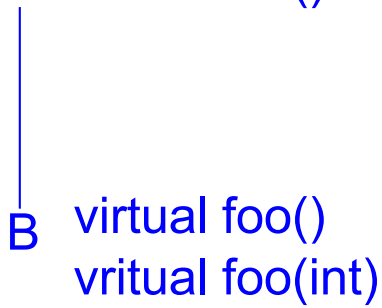
- Java is said to be type safe while C++ is said to be type unsafe

C++ Is Type Unsafe

Object x = new A();
B q = (B) x;
Int case1 = q.foo();
Compiles.
But throws Exception!

```
//#1
void* x = (void *) new A;
B* q = (B*) x;  //a safe downcast?
int case1 = q->foo()//what happens?

//#2
void* x = (void *) new A;
B* q = (B*) x;  //a safe downcast?
int case2 = q->foo(66); //what happens?
```



`q->foo(66)` is a prohibited application (i.e., application of an operation on a value of the wrong type, i.e., forbidden error). Static type `B* q` "promises" the programmer that `q` will point to a `B` object. However, language does not "honor" this promise…

What Is Type Checking

	statically typed	not statically typed (i.e., dynamically typed)
type safe	ML/Ocaml, Haskell, Java*	Python, Scheme, R, JavaScript
type unsafe	C/C++	Assembly

What Is Type Checking?

- Static typing vs. dynamic typing
  - What are the advantages of static typing?
  - What are the advantages of dynamic typing?

Lecture Outline

- *Types*
- *Type systems*
  - *Type checking*
  - *Type safety*
- *Type equivalence*
- *Types in C*

- *Primitive types*
- *Composite types*

Type Equivalence and Type Compatibility

- We now move into the world of procedural von Neumann languages
  - E.g., Fortran, Algol, Pascal and C
  - Value model
  - Statically typed

Type Equivalence and Type Compatibility

- Questions

`e := expression` ☑ or ☒

Are **`e`** and **`expression`** of "same type"?

`foo(arg1, arg2, …, argN)` ☑ or ☒

Do the types of the arguments "match the types" of the formal parameters?

Type Equivalence

- Focus is on constructive view of types
  - Primitive/simple types: e.g., int, char, bool
  - Composite/constructed types:
    - Constructed by applying type constructors
    - pointer e.g., **pointerTo**(int)
    - array e.g., **arrayOf**(char) or **arrayOf(**char,20) or ...
    - record/struct e.g., **record**(age:int, name:arrayOf(char))
    - union e.g. **union**(int, pointerTo(char))

Type Equivalence

- Two ways of defining type equivalence
  - Structural equivalence: based on “shape”
    - Roughly, two types are the same if they consists of the same components, put together in the same way
  - Name equivalence: based on lexical occurrence of the type definition
    - Strict name equivalence
    - Loose name equivalence

- **T1 x; …**
- **T2 y;**

x = y;

Structural Equivalence

- A type is structurally equivalent to itself

- Two types are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent types (i.e., arguments are structurally equivalent)

Aliased Types

- After type declaration
  - **`type n = T`**
  - or **`typedef T n`** in C
- introduces type aliasing
- A type declaration makes **`n`** an alias of **`T`**. **`n`** and **`T`** are said to be aliased types

type Name = String

type String = [Char]

- A type name **`n`** is said to be structurally equivalent to **`T`**

Structural Equivalence

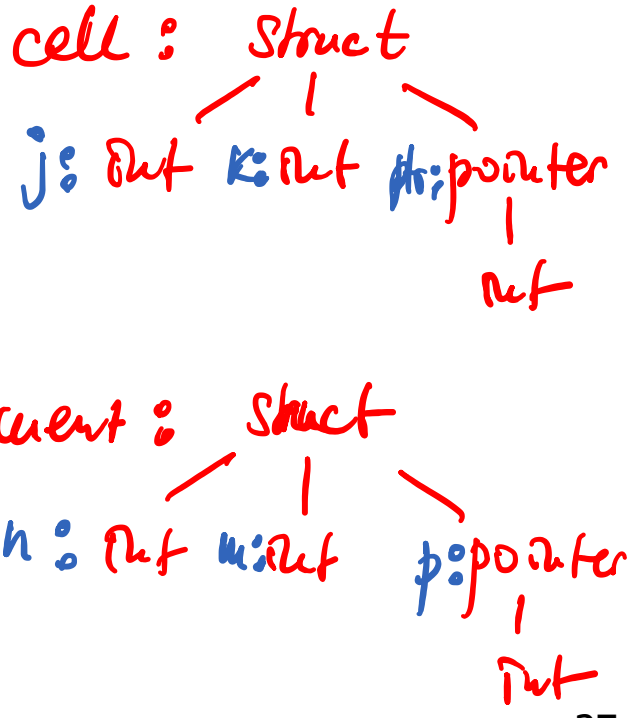
- Example, Pascal-like language:

This is a type definition: an application of the **array** type constructor

```
type S = array [0..99] of char
type T = array [0..99] of char
```

- Example, C:

```
typedef struct {
  int j, int k, int *ptr
} cell;
typedef struct {
  int n, int m, int *p
} element;
```



Structural Equivalence

- Shown by isomorphism of corresponding type trees
  - Show the type trees of these constructed types
  - Are these types structurally equivalent?

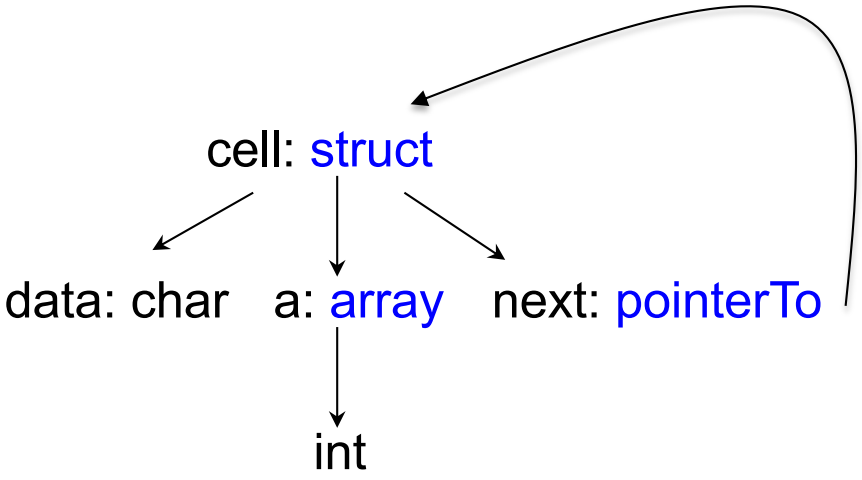
```
struct cell                    struct element
{   char data;                 { char c;
    int a[3];                    int a[5];
    struct cell *next;           struct element *ptr;
}                              }
```

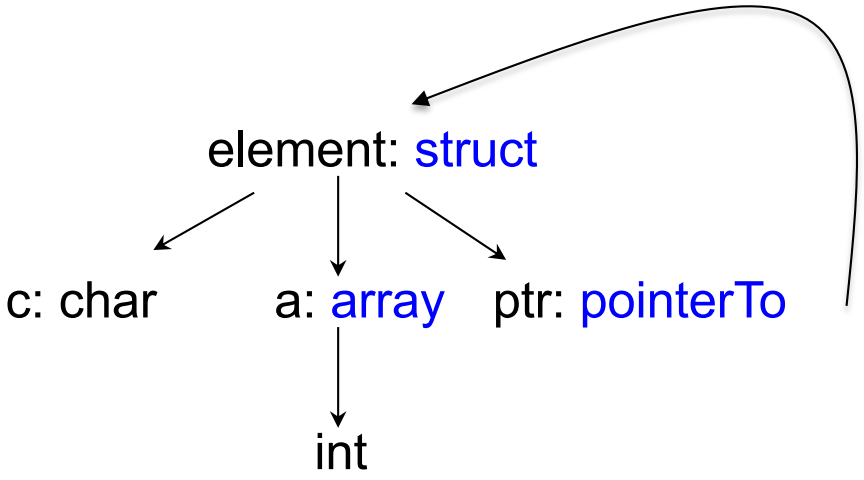
Equivalent types: are field names part of the struct constructed type? YES
are array bounds part of the array constructed type? NO

Structural Equivalence

```
struct cell
{   char data;
    int a[3];
    struct cell *next;
}
```

```
struct element
{   char c;
    int a[5];
    struct element *ptr;
}
```

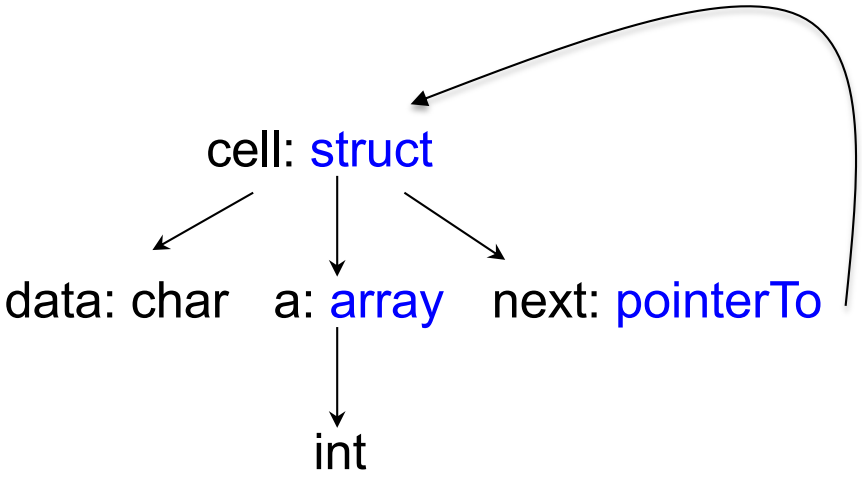


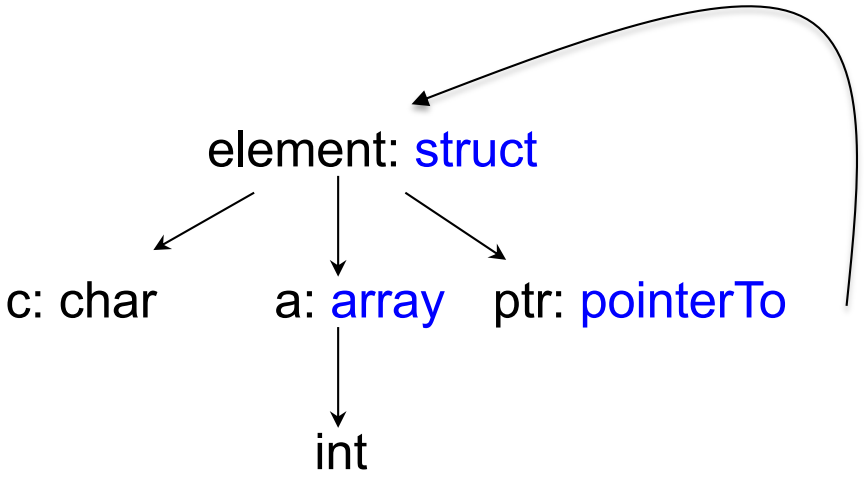


Structural Equivalence

```
struct cell
{  char data;
   int a[3];
   struct cell *next;
}
```

```
struct element
{  char c;
   int a[5];
   struct element *ptr;
}
```





Name Equivalence

Name equivalence

Based on lexical occurrence of type definition. An application of a type constructor is a type definition. E.g., the red `array[1..20]` … is one type definition and the blue `array[1..20]` is a different type definition.

```
type T = array [1..20] of int;
x,y: array [1..20] of int;
w,z: T;
v: T;
```

`x` and `y` are of same type, `w, z, v` are of same type, but `x` and `w` are of different types!

Question

Name equivalence

```
w,z,v: array [1..20] of int;
x,y:   array [1..20] of int;
```

w = z ✓

x = v ✗

Are **x** and **w** of equivalent type according to name equivalence?

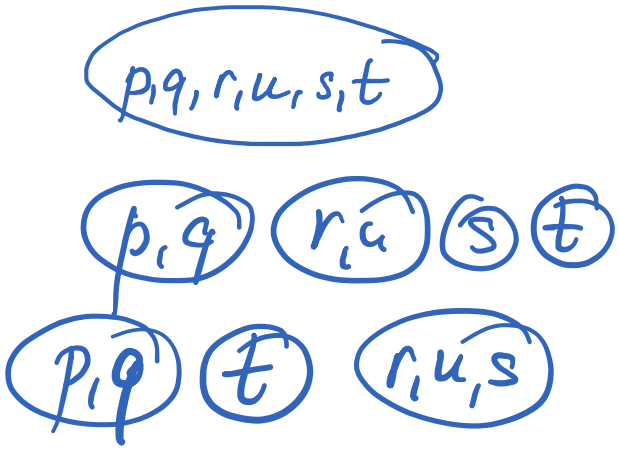
Answer: **x** and **w** are of distinct types.

Name Equivalence

- A subtlety arises with aliased types (e.g., **`type n = T`**, **`typedef int Age`** in C)
- Strict name equivalence
  - A language in which aliased types are considered distinct, is said to have strict name equivalence (e.g., **`int`** and **`Age`** above would be distinct types)
- Loose name equivalence
  - A language in which aliased types are considered equivalent, is said to have loose name equivalence (e.g., **`int`** and **`Age`** would be same)

Exercise

```
type cell = ... // record type
type alink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```



Group `p,q,r,s,t,u` into equiv. classes, according to structural equiv., strict name equiv. and loose name equiv.

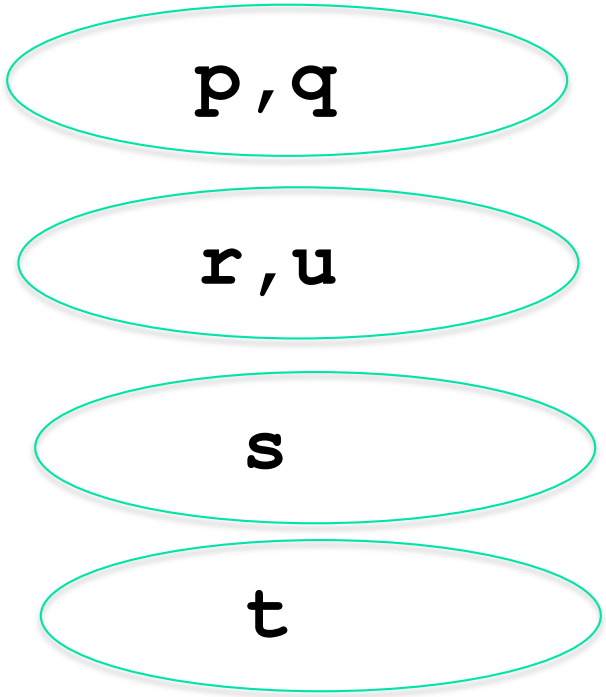
Exercise: Structural Equiv.

```
type cell = ... // record type
type alink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```

p,q,r,s,t,u

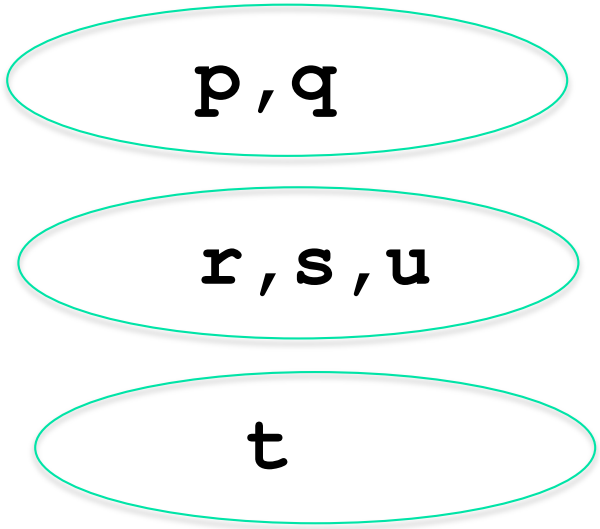
Exercise: Strict Name Equiv.

```
type cell = ... // record type
type alink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```



Exercise: Loose Name Equiv.

```
type cell = ... // record type
type alink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```



The End

- Happy Thanksgiving and Enjoy Your Break!