Types

Read: Scott, Chapters 7.1-7.2 and 8

Announcements

- Check your Rainbow grades
    - Exam 1-2, Quiz 1-7, HW 1-5

- Happy Thanksgiving!

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 set of 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
  - Data abstraction
    - Abstract Data Types (as we saw in Java)
  - Documentation (static types only)

3 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

3 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., **`pointer`**`(int)`
    - array e.g., **`array`**`(char)` or **`array`**`(char,20)` or ...
    - record/struct e.g., **`record`**`(age:int, name:array(char))`
    - union e.g., **`union`**`(int, pointer(char))`

CAN BE NESTED! **`pointer(array(pointer(char)))`**

- For most of us, types are a mixture of these 3 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"

well-typed ill-type

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

What Is a Type System?

1) A set of rules to define syntax of type expressions
2) A set of rules over syntax of language to accept a well-typed program or conversely, reject an ill-typed one (called static semantics)
3) Rules to define semantics of program execution and what it means for the program to "go wrong" (called dynamic semantics)
4) Soundness theorem that links 2) and 3). It states that "well-typed" programs cannot go wrong".

Usually, when we think of a type system, we think of its static semantics, i.e. the rules that accept or reject the program.

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
  - 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 (cdr x)
  - 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?
  - 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?
  - 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?

Type Safety

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

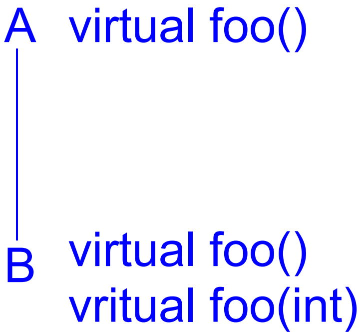
Can we write code that passes the type checker, and yet it calls `quack()` on an object that isn't a `Duck` at runtime?

- In Java?
- In C++?

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

C++ Is Type Unsafe

```
//#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

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

Type Equivalence

- Constructive point of view:
  - Primitive/simple types: e.g., `int, char, bool`
  - Composite/constructed types:
    - Constructed by applying type constructors
    - pointer e.g., `pointer(int)`
    - array e.g., `array(char)` or `array(char,20)` or ...
    - record/struct e.g., `record(age:int, name:array(char))`
    - union e.g., `union(int, pointer(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)
- After type declaration **`type n = T`** or **`typedef T n`** in C, the type name **`n`** is structurally equivalent to **`T`**
  - Declaration makes **`n`** an alias of **`T`**. **`n`** and **`T`** are said to be aliased types

Structural Equivalence

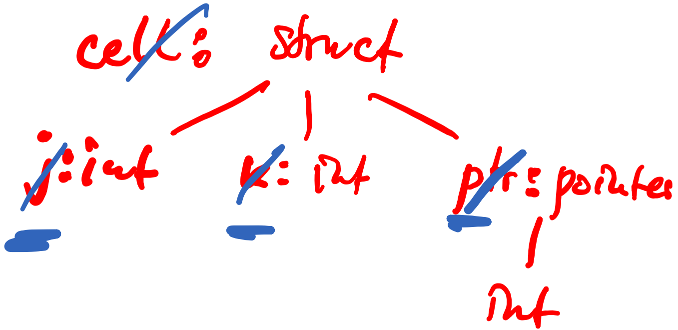
This is a type definition: an application of the **array** type constructor

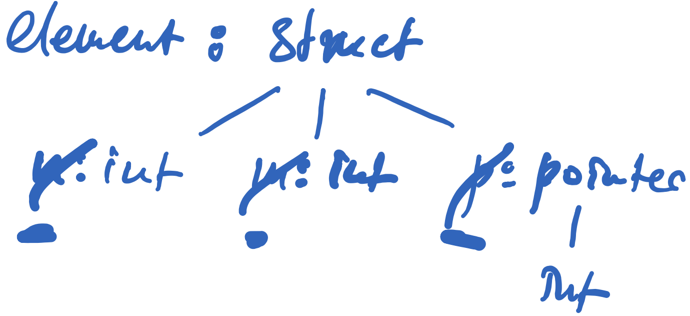
- Example, Pascal-like language:

```
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;
}                           }
```

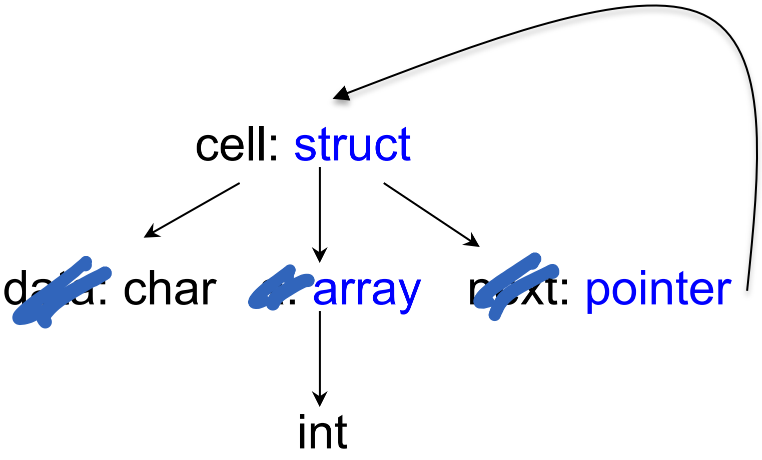
Equivalent types: are field names part of the struct constructed type?
are array bounds part of the array constructed type?

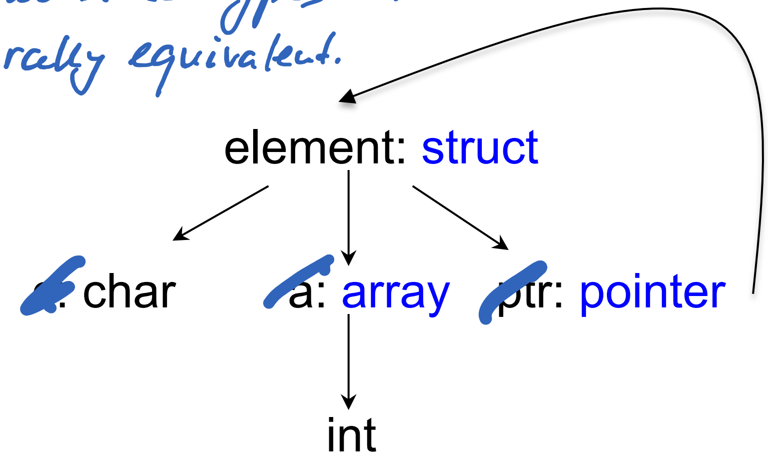
Structural Equivalence

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

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

If field names a not part of the struct constructed type, then yes, two struct types are structurally equivalent.



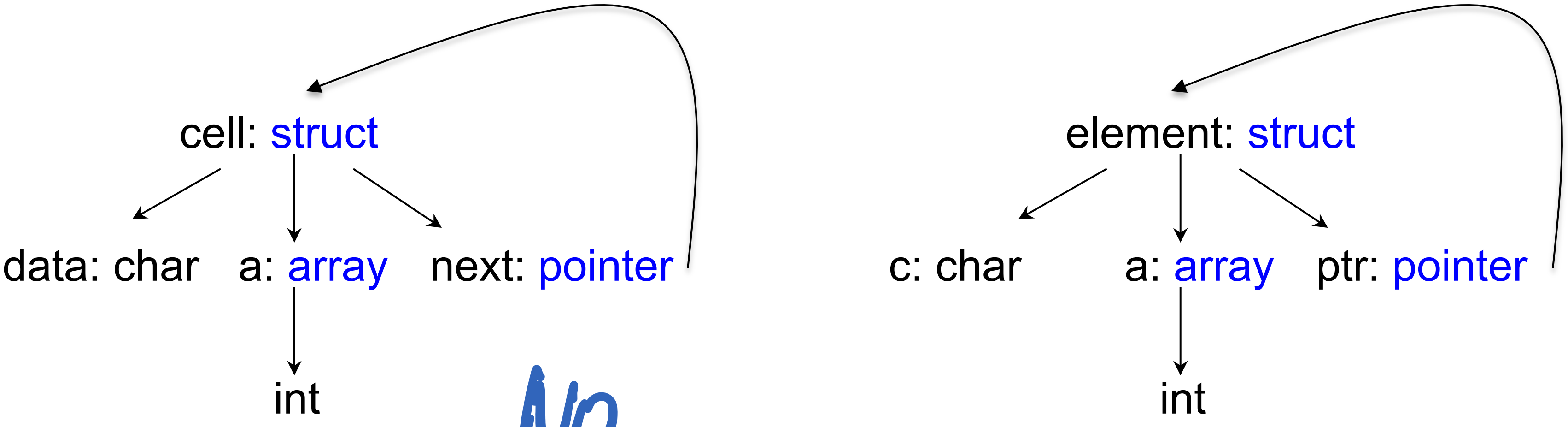


Structural Equivalence

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

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

If field names are part of the struct type, then No, these are distinct types.



cell: struct

data: char

a: array

next: pointer

int

element: struct

c: char

a: array

ptr: pointer

int

No

Name Equivalence

Name equivalence

An application of a type constructor is a type definition.

Under name equivalence each type definition is a distinct type. E.g., the red `array[1..20] of int;` is one type definition (and one type) and the blue `array[1..20] of int;` is a different type definition (and a different type):

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

BLUE array

RED array

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

ANON1

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

ANON2

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`** are 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`** are 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 Equivalence

```
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 Equivalence

```
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,u

s   t

p,q

r,u

s

t

Exercise: Loose Name Equivalence

```
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,u

t

Example: Type Equivalence in C

- First, in the Algol family, field names are part of the record/struct constructed type. E.g., the record types below are NOT even structurally equivalent

```
type A = record
  x,y : real
end;
type B = record
  z,w : real
end;
```

Type Equivalence in C

- Compiler assigns internal (compiler-generated) names to anonymous types

This **struct** is of type anon1.

```
struct RecA          typedef struct       struct
{ char x;            { char x;            { char x;
  int y;               int y;               int y;
} a;                 } RecB;              } c;

RecB b;
```

What variables are of **equivalent type** according to the rules in C?

Type Equivalence in C

- C uses structural equivalence for everything, except unions and structs, for which it uses loose name equivalence

```
struct A       struct B
{ char x;      { char x;
  int y;         int y;
}              }
typedef struct A C;
typedef C *P;
typedef struct B *Q;
typedef struct A *R;
typedef int Age;
typedef int (*F) (int);
typedef Age (*G) (Age);
```

Type Equivalence in C

```
struct B { char x; int y; };
typedef struct B A;
struct { A a; A *next; } aa;
struct { struct B a; struct B *next; } bb;
struct { struct B a; struct B *next; } cc;

A a;
struct B b;

a = b;
aa = bb;
bb = cc;
```

Which of the above assignments pass the type checker?

Question

- Structural equivalence for record types is considered a bad idea. Can you think of a reason why?

Lecture Outline

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

- Primitive types
- Composite types