Announcements

- Quiz 7
- HW 9 is due on Friday
- Rainbow grades
  - HW 1-6 plus 8. Please, read our comments on 8!
  - Exam 1-2
  - Quiz 1-6
- Any questions/concerns, let us know ASAP

Last Class

- Haskell
  - Syntax
  - Lazy evaluation
  - Static typing and type inference
  - Algebraic data types
  - Pattern matching
  - Type classes
  - Monads … and there is a lot more!

Lecture Outline

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

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 (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 = \{1,2,...\}
    - 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

- **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 3 views

---

What Is a Type System?

- A mechanism to define types and associate them with programming language constructs
  - Deduce types of 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 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&lt;A&gt; 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
  - **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], i is of size Bound, i<0 or Bound ≤ i
  - In C, C++, this is not a forbidden error
  - Either 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
    - Either 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
    - Either 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 calls quack() on an object that isn't a Duck?
    - In Java?
    - In C++?
  - Java is said to be type safe while C++ is said to be type unsafe

C++ Is Type Unsafe

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

```c
// #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 A* q “promises” the programmer that q will point to a A object. However, language does not “honor” this promise...

What Is Type Checking

<table>
<thead>
<tr>
<th>statically typed (i.e., dynamically typed)</th>
<th>not statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>type safe</td>
<td>type unsafe</td>
</tr>
<tr>
<td>Assembly</td>
<td></td>
</tr>
</tbody>
</table>
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

Type Equivalence and Type Compatibility

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

Questions

- \( e \ := \ expression \)
  - Are \( e \) and \( expression \) of “same type”?
  - \( a + b \)
    - Are \( a \) and \( b \) of “same type” and type supports +?
  - \( \text{foo}(\text{arg1}, \text{arg2}, \ldots, \text{argN}) \)
    - Do the types of the arguments “match the types” of the formal parameters?

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

Structural Equivalence

- A type name 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 \( \text{type } n = T \) or \( \text{typedef } T \ n \) 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

- Example, Pascal-like language:
  ```
type S = 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;
```

- Shown by isomorphism of corresponding type trees

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

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

Name Equivalence

- Roughly, 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;
```

  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 = ...
```
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:

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

Type Equivalence in C

- Anonymous types are differentiated by internal (compiler-generated) type names:

  ```c
  struct RecA
typedef struct RecB;
  
  RecB b;
  ```

- Which variables are of equivalent type, according to the rules in C?

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

  ```c
  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

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

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

- Implicit conversion — coercion
  - Conversion done implicitly by the compiler
  - In C, mixed mode numerical operations
    - In `e = expression if e is a double and expression is an int, expression is implicitly coerced in to a double`
    - `double d,e;... e = d + 2;` //2 coerced to 2.0

- `int to double,`
- `float to double`
- `How about float to int?`
  - No. May lose precision and thus, cannot be coerced!

Lecture Outline

- Types
  - Type systems
  - Type checking
  - Type safety

- Type equivalence

- Types in C

Pointers: Pointers and Arrays in C

- Pointers and arrays are interoperable:

```
int n;
int *a[n];
int b[10];
```

1. `a = b;`
2. `n = a[3];`
3. `n = *(a+3);`
4. `n = b[3];`
5. `n = *(b+3);`

Type Declaration in C

- What is the meaning of the following declaration in C? Draw the type trees.

1. `int *a[n]`
2. `int (*a)[n]`
3. `int (*f)(int)`
Type Declarations in C

Type tree for PFB:

\[ \text{pointerTo} \rightarrow (\text{int}) \]

Type tree for type of find_p_func:

English: a function that takes a pointer to char as argument, and returns a pointer to a function that takes void as argument and returns int.

Exercise

```c
struct _chunk {
  char name[10];
  int id;
};

struct obstack {
  struct _chunk *chunk;
  struct _chunk *(*chunkfun)();
  void (*freefun)();
};

void chunk_fun(struct obstack *h, void *f) {
  h->chunkfun = (struct _chunk * (*)(())) f;
}

void free_fun(struct obstack *h, void *f) {
  h->freefun = (void (*)()) f;
}

int main() {
  struct obstack h;
  chunk_fun(&h,&xmalloc);
  free_fun(&h,&xfree);
  ...
}
```

Type Declarations in C

Type tree for type of field chunkfun:

\[ \text{pointerTo} \rightarrow \text{pointerTo} \rightarrow \text{pointerTo} \rightarrow (\text{char}) \rightarrow (\text{int}) \]