Administrative Issues

- Midterm Exam (in class)
  Tuesday, Oct. 16th

- Schedule Change
  - HW1
    - HW1-part1 & HW1-part2
    - Due at the same time, as a single file
  - HW2 split into HW2 & HW3
  - Only 1 essay

Academic Dishonesty

- Plagiarism: unacknowledged incorporation of another’s work into work which the student offers for credit.
- Not your words or ideas?
  - You must cite and acknowledge the source!
- Not your source codes?
  - You must not copy it!

How do we detect plagiarism?

- You files will be checked.
  - If they are from WWW/Google:
    - If you can find it, chances are we can find it as well.
    - If they are from other students:
      - We use software to compare students’ files and source codes.
  - We won’t tolerate it!

Type System

- Type Constructors:
  - Build new data types upon simple data types

- Type Checking: The translator checks if data types are used correctly.
  - Type Inference: Infer the type of an expression, whose data type is not given explicitly.
    - e.g., x/y
  - Type Equivalence: Compare two types, decide if they are the same.
    - e.g., x/y and z
  - Type Compatibility: Can we use a value of type A in a place that expects type B?
    - Nontrivial with user-defined types and anonymous types

Strongly-Typed Languages

- Strongly-typed (Ada, ML, Haskell, Java, Pascal)
  - Most data type errors detected at translation time
  - A few checked during execution and runtime error reported (e.g., subscript out of array bounds).

- Pros:
  - No data-corrupting errors can occur during execution. (i.e., no unsafe program can cause data errors.)
  - Efficiency (in translation and execution.)
  - Security/reliability

- Cons:
  - May reject safe programs (i.e., legal programs is a subset of safe programs)
  - Burden on programs, may often need to provide explicit type information.
Weakly-typed and untyped languages

• Weakly-typed: C/C++
  – e.g., interoperability of integers, pointers, arrays.

• Untyped (dynamically typed) languages: scheme, smalltalk, perl
  – Doesn’t necessarily result in data errors.
  – All type checking performed at execution time.
  – May produce runtime errors too frequently.

Security vs. flexibility

• Strongly-typed:
  – No data errors caused by unsafe programs.
  – Maximum restrictiveness, static type checking, illegal safe programs,
    large amount of type information supplied by programmers.

• Untyped:
  – Runtime errors, no data-corruptions. Legal unsafe programs.
  – reduce the amount of type information the programmer must supply.

Safe vs. Legal

Programs

Legal programs

Safe programs

Type Equivalence

• How to decide if two types are the same?

  • Structural Equivalence
    – Types are sets of values
    – Two types are equivalent if they contain the same values.

  • Algol60, FORTRAN

Structural Equivalence

• In C:
  ```c
  struct RecA {
    char x;
    int y;
  };
  struct RecB {
    char x;
    int y;
  };
  struct RecC {
    char x;
    int y;
  };
  struct RecD {
    int y;
    char x;
  };
  {
    struct RecA *a;
    struct RecB *b;
    a = b;  // Warning: incompatible pointer type
  }
  ```

But are they equivalent in these languages?
But are they equivalent in these languages?

- In Java:
  ```java
  class A {
    char x;   int y;
  }
  class B {
    char x;   int y;
  }
  A a = new B();
  ```

Equivalence Algorithm

- If Structural Equivalent is applied:
  ```c
  struct RecA {
    char x;   int y;
  }
  struct RecB {
    char x;   int y;
  }
  struct RecA a;
  struct RecB *b;
  b = &a;
  ```

- Simply ignore the names
  ```c
  struct RecA {
    char x;   int y;
  }
  typedef struct RecA RecA;
  typedef struct {
    char x;   int y;
  } RecB;
  RecB b;
  struct {
    char x;   int y;
  } c;
  ```

Replacing the names by declarations

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

Replacing the names by declarations?

```c
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;
```

```c
struct CharListNode {
    char data;   CharList next;
};
struct CharListNode2 {
    char data;   CharList2  next;
};
```

Cannot do that for recursive types

```c
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;
struct CharListNode {
    char data;   struct CharListNode* next;
};
struct CharListNode2 {
    char data;   struct CharListNode2* next;
};
```
Structural Equivalence

- Can be complicated when they are names, anonymous types, and recursive types
- Simpler, and more strict rules: name equivalence

Name Equivalence

```c
struct RecA { char x; int y; 
typedef struct RecA RecA;
struct RecA *a;
RecA *b;
struct RecA c;
struct { char x; int y; } d;
Struct { char x; int y; } e,f;
a=&c; // ( ok )
a=&d; // (Warning: incompatible pointer type)
b=&d; // (Warning: incompatible pointer type)
a=b; // ( ok, typedef doesn’t create new name )
```

Type Equivalence in C

- Name Equivalence: `struct, union`
- Structural Equivalence: everything else – `typedef` doesn’t create a new type

Example

```c
struct A { char x; int y; 
struct B { char x; int y; }
struct { char x; int y; } C;
typedef struct A * D;
typedef struct A B;
typedef int S[10];
typedef int T[5];
typedef int Age;
typedef int (*F)(int);
typedef Age (*G)(Age);
```

Type Equivalence in Java

- No `typedef`: so less complicated
- `class/interface`: new type (name equivalence, class/interface names)
- arrays: structural equivalence

Type Checking

- **Type Checking**: Determine whether the program is correct in terms of data types.
  - **Type Inference**: Types of expressions
  - **Type Equivalence**: Are two types the same?
  - **Type Compatibility**: Relaxing exact type equivalence under certain circumstances
Example

long y;
float x;
double c;
x = y/2 + c;
• y long, 2 is int, so promoted to long, y/2 long.
• c is double, y/2 is long, so promoted to double, y/2 + c is double.
• x is float, y/2 + c is double, what happens?
  – C?
  – Java?

Example: C

struct RecA {int i; double r;};
int p(struct {int i; double r;} x) {
 ...
}
int q(struct RecA x) {
 ...
}
struct RecA a;
float b;
b = p(a);
b = q(a);

Type Conversion

• Use code to designate conversion?
  – No: automatic/implicit conversion
  – Yes: manual/explicit conversion
• Data representation changed?
  – No, just the type.
  – Yes

Example: Java

• Implicit conversion:
  – Representation change (type promotion, e.g., int to double)
  – No representation change (upcasting)
• Explicit conversion:
  – Representation change (double x = 1.5; Math.round(x))
  – No representation change (downcasting)

Casting in Java

class A {public int x;}
class SubA extends A { public int y;}
A a1 = new A();
A a2 = new A();
SubA suba = new SubA();

al = suba; OK (upcasting)
suba = (SubA) al; OK (downcasting)
suba = a2; compilation error
suba = (SubA) a2; compiles OK, runtime error
al.y; compiles OK, runtime error
if (al instanceof SubA) { ((SubA) al).y; } OK