Function Type in C

typedef int (*IntFunction)(int);

int square(int x) {return x*x;}

IntFunction f = square;

int evaluate(IntFunction g, int value){
    return g(value);
}

printf("%d\n", evaluate(f,3));

Function Type in ML

type IntFunction = int -> int;

fun square(x: int) = x * x;

val f = square;

Fun evaluate(g:IntFunction, value:int) = g value;

evaluate(f,3);

Vector, List

Functional languages:

• Vectors: like arrays, more flexibility, especially dynamic resizability.

• Lists: like vectors, can only be accessed by counting down from the first element.

Pointer

• A pointer type is a type in which the range of values consists of memory addresses and a special value, nil (or null)

• Advantages:
  – Addressing flexibility (address arithmetic, explicit dereferencing and address-of, domain type not fixed [void *])
  – Dynamic storage management
  – Recursive data structures
    • E.g., linked list
      struct CharListNode
      { char data;
        struct CharListNode* next;
      };
      Typedef struct CharListNode* CharList;

Problems with Pointers

• Alias (with side-effect)
  int *a, *b;
  a=(int *) malloc(sizeof(int));
  *a=2;
  b=(int *) malloc(sizeof(int));
  *b=3;
  b=a;
  *b=4;
  printf("%d\n", *a);
Problems with Pointers

- Dangling pointers (dangerous)
  ```c
  int *a, *b;
  a = (int *) malloc(sizeof(int));
  *a = 1;
  b = a;
  free(a);
  printf("%d\n", *b);
  ```

- Garbages (waste of memory)
  ```c
  memory leakage
  int *a;
  a = (int *) malloc(sizeof(int));
  *a=2;
  a = (int *) malloc(sizeof(int));
  ```

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 programmers, 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.
Security vs. flexibility

- Strongly-typed:
  - A type system tries to maximize both flexibility and security, where flexibility means: reduce the number of safe illegal programs & reduce the amount of type information the programmer must supply.
  - Flexibility, no explicit typing or static type checking vs.
  - Maximum restrictiveness, static type checking

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.
- Name Equivalence

Structural Equivalence

- struct RecA {
  char x;
  int y;
}
- struct RecB {
  char x;
  int y;
}
- struct RecC {
  char u;
  int v;
}
- struct RecD {
  int y;
  char x;
}

But are they equivalent in these languages?

- In C:
  struct RecA {
    char x; int y;
  };
  struct RecB {
    char x; int y;
  };
  struct RecA a;
  struct RecB b;
  b=a;  \( \text{(Error: incompatible types in assignment)} \)

- In C:
  struct RecA {
    char x; int y;
  };
  struct RecB {
    char x; int y;
  };
  struct RecA a;
  struct RecB b;
  b=a;  \( \text{(Warning: incompatible types in assignment)} \)
But are they equivalent in these languages?

- In C:
  ```c
  struct RecA {
    char x;   int  y;
  };
  struct RecB {
    char x;   int  y;
  };
  struct RecA a;
  struct RecB* b;
  b = (struct RecB*)&a;  // OK, but does not mean they are equivalent
  ```

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

Equivalence Algorithm

- If structural equivalence is applied:
  ```c
  struct RecA {
    char x;   int  y;
  };
  struct RecB {
    char u;   int  v;
  };
  struct RecA a;
  struct RecB b;
  b = a;
  ```

Replacing the names by declarations

```c
typedef struct { char x; char y } SubRecA;
typedef struct { char x; char y } SubRecB;
```
Cannot do that for recursive types

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

struct CharListNode {
    char data; struct CharListNode* next;
};

struct CharListNode2 {
    char data; struct CharListNode2* next;
};

There are techniques for dealing with this

Name Equivalence

struct RecA { char x; int y; };
typedef struct RecA RecB;
struct RecA *a;
RecB *b;
struct RecA c;
struct { char x; int y; } d;
struct { char x; int y; } e,f;
a=&c; { ok }
b=&d; { Warning: incompatible pointer type }
a=&d; { Warning: incompatible pointer type }
a=b; { ok. Typedef creates alias for existing name }
e=d; { error: incompatible types in assignment }

Type Equivalence in C

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

Example

struct A { char x; int y; };
struct B { char x; int y; };
struct { char x; int y; };
typedef struct A C;
typedef C* P;
typedef struct A * R;
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

Structural Equivalence

• Can be complicated when there are names, anonymous types, and recursive types
• Simpler, and more strict rules: name 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;
int b;
b = p(a);
b = q(a);

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; int y = (int)x)
  - No representation change (downcasting)

Type Conversion

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

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();
a1 = suba; OK (upcasting)
suba = (SubA) a1; OK (downcasting)
suba = a2; compilation error
suba = (SubA) a2;
suba.y; compilation error
if (a1 instanceof SubA) { ((SubA) a1).y; } OK