Announcements

- Homework 7 is due on Thursday, April 21th
  - We have not covered questions 3-5
  - Questions?

- Rainbow grades
  - HW 1-5
  - Exam 1-2
  - Quiz 1-6
  - Any questions/concerns, let us know ASAP

Last Class

- The Applied Lambda Calculus
- The Typed Lambda Calculus
- Types

Lecture Outline

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

Read: Scott, Chapter 7.1 – 7.4

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
    - ADTs
  - 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

3 Views of Types

- **Constructive point of view:**
  - Primitive/simple/builtin 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: string)}`
    - `union e.g., union{(int, pointerTo(char))}`
    - `list e.g., list(...)`
    - `function e.g., float → int`
  - 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

- 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

- **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++
  - 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
  - In C, C++, this is not a forbidden error
  - In Pascal, this is a forbidden error. Prevented with static checks
  - In Java, this is a forbidden error. It is prevented with dynamic checks

- **Type Safety**
  - Java vs C++:
    - 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?

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

virtual foo() is a prohibited application (i.e., 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**

<table>
<thead>
<tr>
<th>statically typed (i.e., dynamically typed)</th>
<th>not statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>type safe ML, Java, Scheme</td>
<td>type unsafe C, C++ Assembly</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?

One more thing...

- What is strong typing?
  - One often hears “Java is strongly typed” while “C++ is weakly typed”...
  - The term is often used in ways that don’t make sense
    - “The language has a type checker”
    - “The language is sound”
    - To most it seems to mean: “A language like C or Java related in a way I can’t make quite precise”
  - Correct (I think): the language has some element of static typing and is type safe

Lecture Outline

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

Type Equivalence and Type Compatibility

- Discussion centers on non-object-oriented, von Neumann (i.e., imperative), statically typed languages: Algol, Pascal and C
- Questions:
  - $e := \text{expression}$ ⊑ or ⊒
    Are $e$ and $\text{expression}$ of “same type”?
  - $a + b$ ⊑ or ⊒
    Are $a$ and $b$ of “same type” and type supports $+$?
  - $\text{foo(arg}_1, \text{arg}_2, \ldots, \text{arg}_N)$ ⊑ or ⊒
    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 type definition
    - Strict name equivalence
    - Loose name equivalence

Structural Equivalence

- A 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:
  
  ```pascal
  type S = array [0..99] of char
  type T = array [0..99] of char
  ```

- Example, C:
  ```c
  typedef struct {
    int j, int k, int *ptr
  } cell;
  typedef struct {
    int n, int m, int *p
  } element;
  ```

Name Equivalence

- **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.

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

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

Question

- **Name equivalence**
  ```c
  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?

Name Equivalence

- A subtlety arises with aliased types (e.g.,
  ```c
  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/struct type**
  ```c
  type cell = ...
  ```

- **Type alink = pointer to cell**
  ```c
  type alink = pointer to cell
  ```

- **Type blink = alink**
  ```c
  type blink = alink
  ```

- **Group p,q,r,s,t into equiv. classes, according to structural equiv., strict name equiv. and loose name equiv.**
  ```c
  p,q : pointer to cell
  r : alink
  s : blink
  t : pointer to cell
  u : alink
  ```
Example: Type Equivalence in C

- First, in the Algol family, field names are part of the 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 { char x, int y; }
typedef struct RecA a;

typedef struct RecB { int y; } 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 { char x; 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

- Which of the above assignments pass?

```c
struct B { char x; int y; };
typedef struct B A;

typedef struct { A a; A *next; } aa;
typedef { struct B a; struct B *next; } bb;
typedef { struct B a; struct B *next; } cc;
```

Which of the above assignments pass?

Question

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

Type Equivalence and Type Compatibility

- Questions:
  - Are e and expression of “same type”?

- e and expression may not be of equivalent types, but they may be of “compatible types”. It may be possible to convert the type of expression to the type of e
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 into a double
    - `double d,e; e = d + 2;` // `e` coerced to 2.0

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

Type Conversion

- Explicit conversion
  - Programmer must “acknowledge” conversion
  - In Pascal, `round` and `trunc` perform explicit conversion
    - `round(s)` real to int by rounding
    - `trunc(s)` real to int by truncating

- In C, type casting performs explicit conversion
  - `freelist *s; ... (char *)s;` forces `s` to be considered as pointing to a char for the purposes of pointer arithmetic

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:

  ```c
  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:**

- `pointerTo` → `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.

```c
Type Declarations in C

- Type tree for type of `chunkfun`:
  - `pointerTo` → `pointerTo`
  - `pointerTo` → `char`
  - `pointerTo` → `int`
  - `struct _chunk` → `struct id: int` → `array char`
```

### Exercise

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

struct obstack {
structure _chunk *chunk;
structure _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,&xfmalloc);
    free_fun(&h,&xfree); ...}
```