CSCI-GA.2110-001 Programming Languages
Spring 2012
Lecture 7

Review

Last week

• Programming Assignment 1
• Types

Outline

• Types (continued)

Sources:

PLP, 7

Type Systems

A type system consists of:

• a mechanism for defining types and associating them with language constructs

• a set of rules for:
  – type equivalence: when do two objects have the same type?
  – type compatibility: where can objects of a given type be used?
  – type inference: how do you determine the type of an expression from the types of its parts
Type Compatibility

Most languages do not require type equivalence in every context. Instead, the type of a value is required to be compatible with the context in which it is used.

What are some contexts in which type compatibility is relevant?

- **assignment statement**: type of lhs must be compatible with type of rhs
- **built-in functions** like +: operands must be compatible with integer or floating-point types
- **subroutine calls**: types of actual parameters (including return value) must be compatible with types of formal parameters

Type Coercion

Coercion in C

The following types can be freely mixed in C:

- **char**
- **(unsigned) (short, long) int**
- **float, double**

Recent trends in type coercion:

- **static typing**: stronger type system, less type coercion
- **user-defined**: C++ allows user-defined type coercion rules
Overloading and Coercion

Overloading: Multiple definitions for the same name, distinguished by their types.

Overload resolution: Process of determining which definition is meant in a given use.

- Usually restricted to functions
- Usually only for static type systems
- Related to coercion. Coercion can be simulated by overloading (but at a high cost). If type \( a \) has subtypes \( b \) and \( c \), we can define three overloaded functions, one for each type. Simulation not practical for many subtypes or number of arguments.

Overload resolution based on:

- number of arguments
- argument types
- return type

Overloading and Coercion

What's wrong with this C++ code?

```cpp
void f(int x);
void f(string *ps);

f(NULL);
```

Depending on how \texttt{NULL} is defined, this will either call the first function (if \texttt{NULL} is defined as 0) or give a compile error (if \texttt{NULL} is defined as \texttt{((void*)0)}).

This is probably not what you want to happen, and there is no easy way to fix it. This is an example of \textit{ambiguity} resulting from coercion combined with overloading.

Overloading and Coercion

What's wrong with this C++ code?

```cpp
void f(int x);
void f(string *ps);

f(NULL);
```

Depending on how \texttt{NULL} is defined, this will either call the first function (if \texttt{NULL} is defined as 0) or give a compile error (if \texttt{NULL} is defined as \texttt{((void*)0)}).

This is probably not what you want to happen, and there is no easy way to fix it. This is an example of \textit{ambiguity} resulting from coercion combined with overloading.

There are other ways to generate ambiguity:

```cpp
void f(int);
void f(char);
double d = 6.02;
f(d);
```
Generic Reference Types

An object of *generic reference type* can be assigned an object of any reference type.

- *void* * in C and C++
- *Object* in JAVA

*How do you go back to a more specific reference type from a generic reference type?*

- Use a type cast
- Some languages include a tag indicating the type of an object as part of the object representation (JAVA, C#, MODULA-3, C++)
- Others (such as C) simply have to settle for unchecked type conversions

Type Inference

*How do you determine the type of an arbitrary expression?*

Most of the time it’s easy:

- the result of built-in operators (i.e. arithmetic) usually have the same type as their operands
- the result of a comparison is Boolean
- the result of a function call is the return type declared for that function
- an assignment has the same type as its left-hand side
Type Inference

How do you determine the type of an arbitrary expression?

Most of the time it’s easy:

- the result of built-in operators (i.e. arithmetic) usually have the same type as their operands
- the result of a comparison is Boolean
- the result of a function call is the return type declared for that function
- an assignment has the same type as its left-hand side

Some cases are not so easy:

- operations on subranges
- operations on composite types

Type Inference for Subrange Operations

Consider this code:

```pascal
type Atype = 0..20;
Btype = 10..20;
var a : Atype;
  b : Btype;
```

What is the type of `a + b`?

- Cheap and easy answer: base type of subrange, integer in this case
- More sophisticated: use bounds analysis to get `10..40`

Type Inference for Subrange Operations

Consider this code:

```pascal
type Atype = 0..20;
Btype = 10..20;
var a : Atype;
  b : Btype;
```

What if we assign to `a` an arbitrary integer expression?
Type Inference for Subrange Operations

Consider this code:

```pascal
type Atype = 0..20;
    Btype = 10..20;
var a : Atype;
    b : Btype;
```

What if we assign to `a` an arbitrary integer expression?

- Bounds analysis might reveal it's OK (i.e. `(a + b) / 2`)
- However, in many cases, a run-time check will be required
- Assigning to some composite types (arrays, sets) requires similar run-time checks

Records

A record consists of a set of typed fields.

Choices:

- **Name or structural equivalence?**
  Most statically typed languages choose name equivalence
  ML, HASKELL are exceptions
- **Nested records allowed?**
  Usually, yes. In FORTRAN and LISP, records but not record declarations can
  be nested
- **Does order of fields matter?**
  Typically, yes, but not in ML
- **Any subtyping relationship with other record types?**
  Most statically typed languages say no. Dynamically typed languages implicitly say yes.
  This is known as *duck typing*:
    - if it walks like a duck and quacks like a duck, I would call it a duck
      - James Whitcomb Riley

Records: Syntax

**PASCAL:**

```pascal
type element = record
    name : array[1..2] of char;
    atomic_number : integer;
    atomic_weight : real;
end;
```

**C:**

```c
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
};
```

**ML:**

```ml
type element = {
    name: string,
    atomic_number: int,
    atomic_weight: real
};
```

Records: Memory Layout

The order and layout of record fields in memory are tied to implementation trade-offs:

- **Alignment** of fields on memory word boundaries makes access faster, but may introduce holes that waste space
- If holes are forced to contain zeroes, comparison of records is easier, but zeroing out holes requires extra code to be executed when the record is created
- Changing the order of fields may result in better performance, but predictable order is necessary for some systems code
Variant Records

A variant record is a record that provides multiple alternative sets of fields, only one of which is valid at any given time.

Each set of fields is known as a variant.

Because only one variant is in use at a time, the variants can share space.

In some languages (e.g. ADA, PASCAL) a separate field of the record keeps track of which variant is valid.

In this case, the record is called a discriminated union and the field tracking the variant is called the tag or discriminant.

Without such a tag, the variant record is called a nondiscriminated union.

Variant Records in Ada

Need to treat group of related representations as a single type:

```ada
type Figure_Kind is (Circle, Square, Line);
type Figure (Kind: Figure_Kind := Square) is record
  Color: Color_Type;
  Visible: Boolean;
  case Kind is
  when Line => Length: Integer;
             Orientation: Float;
             Start: Point;
  when Square => Lower_Left, Upper_Right: Point;
  when Circle => Radius: Integer;
             Center: Point;
  end case;
end record;
```

```ada
C1: Figure(Circle); -- discriminant cannot change
S1: Figure; -- discriminant can change
...
C1.Radius := 15;
if S1.Lower_Left = C1.Center then ...

function Area (F: Figure) return Float is
  -- applies to any figure, i.e., subtype
begin
  case F.Kind is
  when Circle => return Pi * Radius ** 2;
  ...
end Area;
```

Variant Records in Ada

```ada
L : Figure(Line);
S : Figure(Square);
C : Figure; -- defaults to Square
P1 := Point;
...
C := (Circle, Red, False, 10, P1);
  -- record aggregate: C is now a Circle
  ...
C.Orientation ...
  -- illegal, circles have no orientation
C := L;
  -- C is now a line
S := L;
  -- illegal, S cannot change from Square
C.Kind := Square;
  -- illegal, discriminant can only be
  -- changed by assigning whole record
```
Nondiscriminated Unions

Nondiscriminated or free unions can be used to bypass the type model:

```c
union value {
    char *s;
    int i;       // s and i allocated at same address
};
```

Keeping track of current type is programmer’s responsibility.

Can use an explicit tag if desired:

```c
struct entry {
    int discr;
    union {
        // anonymous component, either s or i.
        char *s; // if discr = 0
        int i;   // if discr = 1, but system won’t check
    };
};
```

Note: no language support for safe use of variant!

Discriminated Unions and Dynamic Typing

In dynamically-typed languages, only values have types, not names.

```scheme
(define S 13.45) ; a floating-point number
...
(define S '(1 2 3 4)) ; now it’s a list
```

Run-time values are described by discriminated unions. Discriminant denotes type of value.

Arrays

An array is a mapping from an index type to an element or component type.

- index types
  most languages restrict to an integral type
  ADA, PASCAL, HASKELL allow any scalar type

- index bounds
  many languages restrict lower bound:
  C, JAVA: 0, FORTRAN: 1, ADA, PASCAL: no restriction

- when is length determined
  FORTRAN: compile time; most other languages: can choose

- dimensions
  some languages have true multi-dimensional arrays (FORTRAN)
  most simulate multi-dimensional arrays quite nicely as arrays of arrays.

- first-classness
  C, C++ do not allow arrays to be returned from functions

- a slice or section is a rectangular portion of an array
  Some languages (e.g. FORTRAN, PERL, PYTHON, APL) have a rich set of
  array operations for creating and manipulating sections.

Array Literals

- ADA: (23, 76, 14)
- SCHEME: #(23, 76, 14)
- C and C++ have initializers, but not full-fledged literals:
  ```c
  int v2[] = { 1, 2, 3, 4 }; // size from initializer
  char v3[2] = { 'a', 'z' }; // declared size
  int v5[10] = { -1 };       // default: other components
  struct School r =
    ( "NYU", 10012 );        // record initializer
  char name[] = "Scott";    // string literal
  ```
Array Shape

The *shape* of an array consists of the number of dimensions and the bounds of each dimension in the array.

The time at which the shape of an array is bound has an impact on how the array is stored in memory:

- **global lifetime, static shape**: static global memory
- **local lifetime, static shape**: part of local stack frame
- **local lifetime, shape bound at runtime**: variable-size part of local stack frame
- **arbitrary lifetime, shape bound at runtime**: allocate from heap or reference to existing array
- **arbitrary lifetime, dynamic shape**: also known as *dynamic arrays*, must allocate (and potentially reallocate) in heap

Array Memory Layout

Two-dimensional arrays

- **Row-major layout**: Each row of array is in a contiguous chunk of memory
- **Column-major layout**: Each column of array is in a contiguous chunk of memory
- **Row-pointer layout**: An array of pointers to rows lying anywhere in memory

If an array is traversed differently from how it is laid out, this can dramatically affect performance (primarily because of cache misses)

A *dope vector* contains the dimension, bounds, and size information for an array. Dynamic arrays require that the dope vector be held in memory during run-time.

Pointers

- **value model** pointer has a value that denotes a memory location (C, PASCAL, ADA)
- **reference model** names have dynamic bindings to objects, pointer is implicit (ML, LISP, SCHEME)
- **JAVA** uses value model for built-in (scalar) types, reference model for user-defined types

```plaintext
type Ptr is access Integer; -- Ada: named type
typedef int *ptr; // C, C++
```

Extra pointer capabilities

Questions:

- Is it possible to get the address of a variable? Convenient, but aliasing causes optimization difficulties. (the same way that pass by reference does)
  Unsafe if we can get the address of a stack allocated variable (allowed in C/C++)
- Is pointer arithmetic allowed? unsafe if unrestricted
  In C/C++, no bounds checking:

```c
// allocate space for 10 ints
typedef int *ptr; // C, C++
int *p = malloc(10 * sizeof(int));
p += 42;
... *p ... // out of bounds, but no check
```
Recursive data structures

```plaintext
type Cell; -- an incomplete type
type Ptr is access Cell; -- an access to it
type Cell is record -- the full declaration
  Value: Integer;
  Next, Prev: Ptr;
end record;
List: Ptr := new Cell’(10, null, null);
... -- A list is just a pointer to its first element
List.Next := new Cell’(15, null, null);
List.Next.Prev := List;
```

Recursive data structures in C++

```plaintext
struct cell {
  int value;
  cell *prev;
  cell *next; // legal to mention name
}; // before end of declaration
struct list; // incomplete declaration
struct link {
  link *succ;
  list *memberOf;
}; // a pointer to it
struct list { // full definition
  link *head; // mutual references
};
```

Pointers and dereferencing

- Need notation to distinguish pointer from designated object
  - in ADA: `Ptr vs Ptr.all`
  - in C: `ptr vs *ptr`
  - in JAVA: no notion of pointer (only references)
- For pointers to composite values, dereference can be implicit:
  - in ADA: `C1.Value` equivalent to `C1.all.Value`
  - in C/C++: `c1.value` and `c1->value` are different

Pointers and arrays in C/C++

In C/C++, the notions:

- an array
- a pointer to the first element of an array

are almost the same.

```plaintext
void f (int *p) { ... }
int a[10];
f(a); // same as f(&a[0])

int *p = new int[4];
... p[0] ... // first element
... *p ... // first element
... p[1] ... // second element
... *(p+1) ... // second element
... p[10] ... // past the end; undetected error
```
### Pointers and safety

Pointers create aliases: accessing the value through one name affects retrieval through the other:

```c
int *p1, *p2;
...
p1 = new int[10]; // allocate
p2 = p1; // share
delete[] p1; // discard storage
p2[5] = ... // error:
    // p2 does not denote anything
```

### Pointer troubles

Several possible problems with low-level pointer manipulation:

- dangling references
- garbage (forgetting to free memory)
- freeing dynamically allocated memory twice
- freeing memory that was not dynamically allocated
- reading/writing outside object pointed to

### Dangling references

If we can point to local storage, we can create a reference to an undefined value:

```c
int *f () { // returns a pointer to an integer
    int local; // variable on stack frame of f
    ...
    return &local; // reference to local entity
}
```

```c
int *x = f ();
...
*x = 5; // stack may have been overwritten
```

### Lists, sets and maps

- list: ordered collection of elements
- set: collection of elements with fast searching
- map: collection of (key, value) pairs with fast key lookup

Low-level languages typically do not provide these. High-level and scripting languages do, some as part of a library.

- PERL, PYTHON: built-in, lists and arrays merged.
- C, FORTRAN, COBOL: no
- C++: part of STL: `list<T>, set<T>, map<K,V>`
- JAVA: yes, in library
- SETL: built-in
- ML, HASKELL: lists built-in, set, map part of library
- SCHEME: lists built-in
- PASCAL: built-in sets
  but only for discrete types with few elements, e.g., 32
Function types

- not needed unless the language allows functions to be passed as arguments or returned
- variable number of arguments:
  - C/C++: allowed, type system loophole, JAVA: allowed, but no loophole
- optional arguments: normally not part of the type.
- missing arguments in call: in dynamically typed languages, typically OK.