6.035

Fall 2000

Lecture 10: Type Checking

Outline

- Program correctness
- Type systems
- Semantics of an object-oriented program
- Polymorphic Types

Overview

- Program Correctness
  - Lots of things can go wrong when a program runs
  - Major theme in CS: minimizing possibility of incorrect execution
- Programming language design impact on which errors are caught and when they are caught
- Good programming language design can minimize (but not eliminate) this possibility
- Trade-off between safety and expressive power

Errors in Principle and Practice

- Extreme View:
  - Each program has a specification
  - If program does not satisfy its specification, it is just wrong, and has no redeeming value
- Problems With This View:
  - What is the specification anyway?
  - Other issues are important
    - time to market, features, performance, effect of errors
  - Especially true for shrink-wrap software with no direct effect on physical world (Microsoft Word)

Safe Versus Unsafe Programming Languages

- Safe Programming Language
  - Every program has a single well-defined result, even if the result is some kind of error
  - Examples: Scheme, Java
- Unsafe Programming Language
  - Some programs are undefined, and ANYTHING can happen when you run them
  - Examples: C, C++, Fortran

Sources of Unsafety

- Explicit Memory Deallocation
  - Dangling References
  - Memory Leaks (not, strictly speaking, unsafe)
  - Safety mechanism: garbage collection
  - Do garbage collected programs have memory leaks?
- Trade-off: Who controls memory management
  - Programmer controls explicit memory deallocation (can turn into speed and memory usage benefits)
  - PL implementation controls garbage collection

More Sources of Unsafety

- Stack allocation and escaping pointers

```
char *translate(char *in, int n) {
   char result[128];
   int i;
   for (i = 0; i < n; i++) {
        result[i] = in[i]+1;
   }
   return(result);
}
```


Eliminating Stack Problems

- Allocate Activation Records In Heap, Not Stack
  - Scheme
  - ML
- Disallow pointers into stack (Java)

More Sources of Unsafety

- Array bounds violations
- Eliminated by explicit array bounds checks
  - But Run-Time Overhead
  - Complicates Implementation of Pointers Into Arrays and Pointer Arithmetic
  - So Java (for example) has array bounds checks but no pointers and pointer arithmetic
- Uninitialized Variables
  - Solution: default values (initialization overhead)

Static Type Safety

- Operations can usually be applied some but not all kinds of data
  - addition - only integers, floating point numbers, etc.
  - field access - must be from object with that field
  - method invocation - must be on object with that method
- Implementation of statically type safe language statically checks that no type violations occur when program runs
- Most common form of static program checking

Type System

- Programmer declares (or system infers) types
  - In Java, declare types of variables
  - In ML, static type inferencer infers types
- Implementation checks that types will not be violated at run time
- Type systems inevitably rule out some programs with no type errors
- Only question is how restrictive they are

10

Outline

- Program correctness
- Type systems
- Semantics of an object-oriented program
- Polymorphic Types

17

Type Systems

- A type system is used for type checking
- A type system incorporates
  – syntactic constructs of the language
  – notion of types
  – rules for assigning types to language constructs

Type expressions

- A compound type is denoted by a type expression
- A type expression is
  – a basic type
  – application of a type constructor to other type expressions

18

Type Expressions: Basic types

- Atomic types defined by the language
- Examples:
  – integers
  – booleans
  – floats
  – characters
- type_error
  – special type that'll signal an error
- void
  – basic type denoting "the absence of a value"

Type Expressions: Names

- Since type expressions maybe be named, a type name is a type expression

Type Expressions: Products

- If T_1 and T_2 are type expressions $T_1 \times T_2$ is also a type expression

19

Type Expressions: Arrays

- If T is a type expressions a **array(T, I)** is also a type expression
  – I is a integer constant denoting the number of elements of type T
  – Example:
    `int foo[128];`
    array(integer, 128)

Type Expressions: Function Calls

- Mathematically a function maps
  - elements of one set (the domain)
  - to elements of another set (the range)
- Example

```
int foobar(int a, boolean b, int c)
```

integer × boolean × integer × integer → integer

Type Expressions: Some others

- Records
  - structures and classes
  - Example

```
class { int i; int j;}
```

integer × integer

- Functional Languages
  - functions that take functions and return functions
  - Example

(integer → integer) × integer → (integer → integer)

A simple typed language

- A language that has a sequence of declarations followed by a single expression

P → D; E

D → D; D | **id** : T

T → **char** | **integer** | **array** [**num**] **of** T

E → **literal** | **num** | **id** | E + E | E [E]

- Example Program

```
var: integer;
var + 1023
```


A simple typed language

- A language that has a sequence of declarations followed by a single expression

P → D; E

D → D; D | **id** : T

T → **char** | **integer** | **array** [**num**] **of** T

E → **literal** | **num** | **id** | E + E | E [E]

- What are the semantic rules of this language?

Parser actions

P → D; E

D → D; D

D → **id** : T { addtype(id.entry, T.type); }

T → char { T.type = char; }

T → integer { T.type = integer; }

T → array [num] of T_1

{ T.type = array(T_1.type, num.val); }

Parser actions

E → literal { E.type = char; }

E → num { E.type = integer; }

E → id { E.type = lookup_type(id.name); }

24

Parser actions

$E \rightarrow E_1 + E_2$ { if E_1.type == integer and
E_2 .type == integer then
E.type = integer
else
E.type = type_error
}

Parser actions

$E \rightarrow E_1 [E_2]$ { if E_2.type == integer and
E_1 .type == array(s, t) then
E.type = s
else
E.type = type_error
}

25

Type Equivalence

- How do we know if two types are equal?
  - Same type entry
  - Example:

```
int A[128];
foo(A);

foo(int B[128]) { … }
```

    - Two different type entries in two different symbol tables
    - But they should be the same

Structural Equivalence

- If the type expression of two types have the same construction, then they are equivalent
- "Same construction"
  - Equivalent base types
  - Same set of type constructors are applied in the same order (i.e. equivalent type tree)

26

Type Coercion

- Implicit conversion of one type to another type
- Example

```
int A;
float B;
B = B + A
```

- Two types of coercion
  - widening conversions
  - narrowing conversions

Widening conversions

- Conversions without loss of information
- Examples:
  - integers to floats
  - shorts to longs

Narrowing conversions

- Conversions that may loose information
- Examples:
  - integers to chars
  - longs to shorts
- Rare in languages

Type casting

- Explicit conversion from one type to another
- Both widening and narrowing
- Example

```
int A;
float B;
A = A + (int)B
```

- Unlimited typecasting can be dangerous

Question:

- Can each variables, functions and operators have a unique type?
- How about +, what is its type?

Overloading

- Some operators may have more than one type.
- Example

```
int A, B, C;
float X, Y, Z;
A = A + B
X = X + Y
```

- Complicates the type system
  - Example

    ```
    A = A + X
    ```

    - What is the type of + ?

Outline

- Program correctness
- Type systems
- Semantics of an object-oriented program
- Polymorphic Types

Class

- A class is an abstract data type
- It contains:
  - Data (fields)
  - Actions (methods)
  - Access restrictions
- Each instance of a class will create a separate object
  - its own copy of instance variables (fields)
  - shares the actions

Example class

```
class vehicle {
    int num_wheels;
    void print_num_wheels( )  { … }
}
```

field

method

```
vehicle A;
A.print_num_wheels( )
```

- Object is an implicit parameter to the method call

Inheritance

- Extends classes by allowing for supertype/subtype relationships
- Supports incremental code reuse
  - common parts in a common supertype
  - individual differences in each subtype

Inheritance example

```
class SUV extends vehicle {
    int rollover_speed;
    int get_rollover_speed( ) { … }
    void print_rollover_speed( ) { … }
}
```

- class SUV is a subclass of the class vehicle
- class vehicle is a superclass of the class SUV
- An instance (object) of class SUV contains
  - all the fields of the class vehicle
  - all the fields of the class SUV
- Methods in both SUV and vehicle are visible to the class SUV

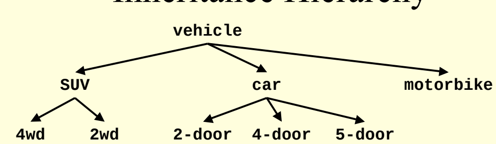
Inheritance

- Single Inheritance
  - when each class is restricted to have at most one immediate superclass
- Multiple Inheritance
  - when each class is permitted to have more than one immediate superclass

Inheritance Hierarchy

- The superclass/subclass relationship
  - defined by the extends
  - can be modeled by a directed acyclic graph (DAG)

Inheritance Hierarchy



- Car is a child of vehicle (immediate subclass)
- vehicle is a parent of SUV (immediate superclass)
- 4wd is a descendant of vehicle (subclass)
- vehicle is a ancestor of 2-door (superclass)

Access Control Rules

- Set of type access rules used by a generic OO language (i.e. espresso)
  - Scope visibility
  - Data access
  - Access to public methods
  - Access to private methods
- Many OO languages have more complicated access controls

Scope visibility

- Variables and fields of a class can be declared anywhere in a program that a declaration is permitted and the class definition is visible.
- If a filed in the subclass and superclass uses the same name
  - name resolution uses scope rules
  - treats subclass's scope within the superclass's scope

Data access

- Data fields of a class can only be accessed by the methods defined in that class
- A more permissive variation:
  - All the methods in the subclasses can access data-fields in the superclass

Access to public methods

- All public methods of a class can be invoked by any method that can declare a variable or a field of the class.

Access to private methods

- Private methods of a class can be only invoked by:
  - methods of that class
  - methods of any class that is a descendant

Example: C++ access controls

- a class can be a *friend* of another class
- methods and fields can be
  - private: visible to member functions and friends
  - protected: visible to member functions and friends and derived classes (and their friends)
  - public: can be used by any function

35

Automatic type conversion

- An expression of a class is coerced into an ancestor class when required
  - but not vice versa
  - called “up-casting”
  - Always legal because subclass contains all the fields of the superclass
- Down-casting
  - this is more permissive
  - explicit conversion from an ancestor class to a descendant class
  - Only meaningful if the object was initially created as in the subclass but later converted to a superclass
  - Cannot be check at compile-time

36

Inheritance vs. Aggregation

- A class T2 is an aggregation of class T1 if class T2 contains one or more fields of type T1
  - Unlike inheritance, T2 cannot access private fields or methods in T1
- When to inherit and when to aggregate
  - inherit: T2 is a T1
  - aggregate: T2 has a T1

Example of Inheritance vs. Aggregation

- SUV is a vehicle
- SUV has an engine

```
class vehicle {
     …
}

class SUV extends vehicle {
     engine power_plant;
     …
}
```


Multiple Inheritance

- Allows a class to be an extension of multiple classes
  - Leads to more complicated semantics for subtyping

Example of Multiple Inheritance

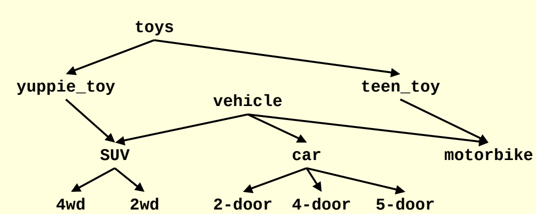
```
class vehicle {
     …
}

class yuppie_toys {
     …
}

class SUV extends vehicle, yuppie_toys {
     …
}
```

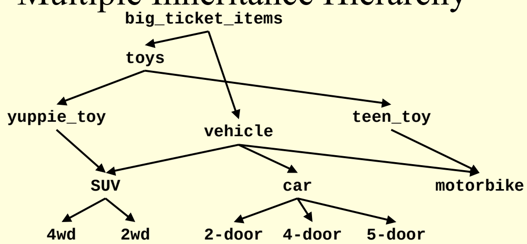

37

Multiple Inheritance Hierarchy



- Inheritance hierarchy is a DAG
- Question: if yuppie_toys and vehicle both has a method price() when SUV invoke price, which method should be invoked?

Multiple Inheritance Hierarchy



- Even more complicated when there is a common ancestor
- Question: How many instances of bti be included in an instance of SUV?

Religious Wars in Single Versus Multiple Inheritance

- Single Inheritance leads to classification hierarchies. Fits in great with one philosophy of object-oriented programming.
- Multiple inheritance leads to composable toolkits and implementation hacks.
- Conceptual problems with multiple inheritance
  - naming conflicts
  - repeated inheritance
- See www.elj.com/eiffel/feature/inheritance/mi/review

Outline

- Program correctness
- Type systems
- Semantics of an object-oriented program
- Polymorphic Types

What is a Polymorphic Type?

- Ordinary procedures allow the body to be executed with arguments of fixed types.
- Each call to a polymorphic procedure executes the body with the types of the arguments
- Benefits of polymorphism
  - Code reuse
  - Example
    - same sort procedure can be applied to a list of integers as well as a list of strings

Parametric Polymorphism

- Procedures have types parameterized
- Instantiate the procedure with a given set of types
- Templates in C++
- Example:

```
template<class T> class linked_list_elem {
  T elem; linked_list_elem * next;
  ...
}
lined_list_elem<int> integer_list;
lined_list_elem<foo> foo_list;
```


Parametric Polymorphism

- Would like to implement container classes once
- But in Java, must cast down when extract something in a container class
- Potential for something like a type error
- Options
  - Everybody in a list inherits from class list
    - But trashes hierarchy, and may be in multiple lists
  - Everybody implements list interface
    - But must reimplement methods all the time, and multiple list problem remains

Stack in Java

```
interface stack {
   public void push(Object x);
   public Object pop();
}
```

- If I build a stack of point objects, have to cast down to point when pop stack
  - point p; stack s; s.push(p); p = (point) s.pop();
- Cast may fail if programmer gets types wrong

GJ - Parametric Polymorphism in Java

```
class stack<T> {
   public void push(T x);
   public T pop();
}
```

- Can now pop stuff off with no casts
  - point p; stack s<point>; s.push(p); p = s.pop();
- GJ implemented as a front end to Java
- Automatically inserts down casts

Dynamic Typing

- Still type safe
- But catch type errors when program runs
  - Scheme, Smalltalk, Self
- Typically more flexible
  - In Smalltalk, can send foo to any object that has foo method. In Java, type system can get in way.

Dynamic versus Static Typing

- Dynamic languages check types only as program runs.
- More flexibility
- But catch all type errors dynamically, not statically.
- Argument over how important it this is.