CS412/413

Introduction to Compilers

Radu Rugina

Lecture 14: Objects

25 Feb 04

Records

- Objects combine features of records and abstract data types
- Records = aggregate data structures
  - Combine several variables into a higher-level structure
  - Type is essentially cartesian product of element types
  - Need selection operator to access fields
  - Pascal records, C structures
- Example: struct {int x; float f; char a,b,c; int y } A;
  - Type: {int x; float f; char a,b,c; int y }
  - Selection: A.x = 1; n = A.y;

ADTs

- Abstract Data Types (ADT): separate implementation from specification
  - Specification: provide an abstract type for data
  - Implementation: must match abstract type
- Example: linked list

implementation

```
Cell = { int data; Cell next; }
List = {int len; Cell head, tail; }
int length() { return l.len; }
int first() { return head.data; }
List rest() { return head.next; }
List append(int d) { … }
```

specification

```
int length();
List append (int d);
int first();
List rest();
```


Objects as Records

- Objects also have fields
- … in addition, they have methods = procedures which manipulate the data (fields) in the object
- Hence, objects combine data and computation

```
class List {
    int len;
    Cell head, tail;
    int length();
    List append(int d);
    int first();
    List rest();
}
```


Objects as ADTs

- Specification: public methods and fields of the object
- Implementation: Source code for a class defines the concrete type (implementation)

```
class List {
    private int len;
    private Cell head, tail;
    public static int length() {…};
    public static List append(int d) {…};
    public static int first() {…} ;
    public static List rest() {…};
}
```


Objects

- What objects are:
  - Aggregate structures which combine data (fields) with computation (methods)
  - Fields have public/private qualifiers (can model ADTs)
- Need special support in many compilation stages:
  - Semantic analysis (type checking!)
  - Analysis and optimizations
  - Implementation, run-time support
- Features:
  - inheritance, subclassing, subtyping, dynamic dispatch

Inheritance

- Inheritance = mechanism which exposes common features of different objects
- Class B extends class A = "B has the features of A, plus some additional ones", i.e., B inherits the features of A
  - B is subclass of A; and A is superclass of B

```
class Point {
        float x, y;
        float getx();
        float gety();
}
class ColoredPoint extends Point {
        int color;
        int getcolor();
}
```


Single vs. Multiple Inheritance

- Single inheritance: inherit from at most one other object (Java)
- Multiple inheritance: may inherit from multiple objects (C++)

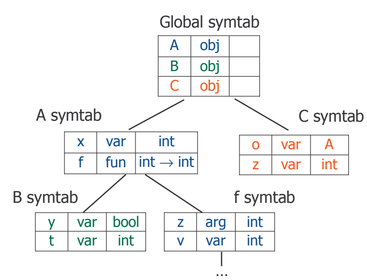
```
class A {
   int a;
   int geta();
}

class B {
   int b;
   int getb();
}

class C : A, B {
   int c;
   int getc();
}
```


Inheritance and Scopes

- How do objects access fields and methods of:
  - Their own?
  - Their superclasses?
  - Other unrelated objects?
- Each class declarations introduces a scope
  - Contains declared fields and methods
  - Scopes of methods are sub-scopes
- Inheritance implies a hierarchy of class scopes
  - If B extends A, then scope of A is a parent scope for B

Example

```
class A {
   int x;
   int f(int z) {
      int v; …
   }
}
class B extends A {
   bool y;
   int t;
}
class C {
   A o;
   int z;
}
```



Class Scopes

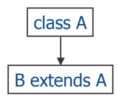
- Resolve an identifier occurrence in a method:
  - Look for symbols starting with the symbol table of the current block in that method
- Resolve qualified accesses:
  - Accesses o.f, where o is an object of class A
  - Walk the symbol table hierarchy starting with the symbol table of class A and look for identifier f
  - Special keyword this refers to the current object, start with the symbol table of the enclosing class

Class Scopes

- Multiple inheritance:
  - A class scope has multiple parent scopes
  - Which should we search first?
  - Problem: may find symbol in both parent scopes!
- Overriding fields:
  - Fields defined in a class and in a subclass
  - Inner declaration shadows outer declaration
  - Symbol present in multiple scopes

Inheritance and Typing

- Classes have types
  - Type is cartesian product of field and method types
  - Type name is the class name
- What is the relation between types of parent and inherited objects?
- Subtyping: if class B extends A then
  - Type B is a subtype of A
  - Type A is a supertype B

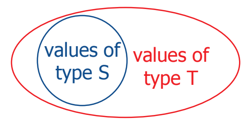


- Notation: B <: A

Subtype ≈ Subset

"A value of type S may be used wherever a value of type T is expected"

S <: T → values(S) $\subseteq$ values(T)



Subtype Properties

- If type S is a subtype of type T (S <: T), then: A value of type S may be used wherever a value of type T is expected (e.g., assignment to a variable, passed as argument, returned from method)

```
Point x;
ColoredPoint y;
x = y;
```

ColoredPoint <: Point
↑ subtype ↑ supertype

- Polymorphism: a value is usable at several types
- Subtype polymorphism: code using T's can also use S's; S objects can be used as S's or T's.

Implications of Subtyping

- We don't actually know statically the types of objects
  - Can be the declared class or any subclasses
  - Precise types of objects known only at run-time
- Problem: overriden fields / methods
  - Declared in multiple classes in the hierarchy
  - We don't know statically which declaration to use at compile time
  - Alternative: use statically declared type (e.g. for fields)
  - For methods we would like the precise object type

Virtual Functions

- Virtual functions = methods overriden by subclasses
  - Subclasses define specialized versions of the methods

```
class List {
  List next;
  int length() { … }
}

class LenList extends List {
  int n;
  int length() { return n; }
}
```


Virtual Functions

- We don't know what code to run at compile time

```
List a;
if (cond)  { a = new List(); }
else       { a = new LenList(); }
a.length()
```

⇒ List.length() or LenList.length() ?

- Solution: method invocations resolved dynamically
- Dynamic dispatch: run-time mechanism to select the appropriate method, depending on the object type

Objects and Typing

- Objects have types
  - … but also have implementation code for methods
- ADT perspective:
  - Specification = typing
  - Implementation = method code, private fields
  - Objects mix specification with implementation
- Can we separate types from implementation?

Interfaces

- Interfaces are pure types; they don't give any implementation

implementation

```
class MyList implements List {
   private int len;
   private Cell head, tail;

   public int length() {…};
   public List append(int d) {…};
   public int first() {…} ;
   public List rest() {…};
}
```

specification

```
interface List {
   int length();
   List append(int d);
   int first();
   List rest();
}
```


Multiple Implementations

- Interfaces allow multiple implementations

```
interface List {
      int length();
      List append(int);
      int first();
      List rest(); }
```

⇨

```
class SimpleList impl. List {
   private int data;
   private SimpleList next;
   public int length()
      { return 1+next.length() } …
```

⇩

```
class LenList implements List {
      private int len;
      private Cell head, tail;
      private LenList() {…}
      public List append(int d) {…}
      public int length() { return len; }
      …
```


Subtyping vs. Subclassing

- Can use inheritance for interfaces
  - Build a hierarchy of interfaces

  interface A {…}
  interface B extends A {…}

  $B <: A$

- Objects can implement interfaces

  class C implements A {…}

  $C <: A$

- Subtyping: interface inheritance
- Subclassing: object (class) inheritance
  - Subclassing implies subtyping

Abstract Classes

- Classes define types and some values (methods)
- Interfaces are pure object types
- Abstract classes are halfway:
  - define some methods
  - leave others unimplemented
  - no objects (instances) of abstract class

Subtypes in Java

interface I_1 extends I_2 { … }	class C implements I { … }	class C_1 extends C_2
I_2 | I_1	I | C	C_2 | C
$I_1 <: I_2$	$C <: I$	$C_1 <: C_2$

Subtyping Properties

- Subtype relation is reflexive: T <: T
- Transitive: R <: S and S <: T implies R <: T
- Anti-symmetric:

$$T_1 <: T_2 \wedge T_2 <: T_1 \Rightarrow T_1 = T_2$$

- Defines a partial ordering on types!
- Use diagrams to describe typing relations

Subtype Hierarchy

- Introduction of subtype relation creates a hierarchy of types: subtype hierarchy