Object Calculus II

CS242
Lecture 8
Review: Record Types

• Conceptually an object is a record of fields and methods
  \[ \text{flag} = \text{False}, \text{value} = 42, \text{add}(i: \text{Int}): \text{Int} \]

• For types we use function types
  \[ \text{flag}: \text{Bool}, \text{value}: \text{Int}, \text{add}: \text{Int} \rightarrow \text{Int} \]
Untyped Object Calculus Syntax

• An object is a finite map from field names to methods that produce objects

\[ o = [ ..., l_i = \zeta(x) \ b_i, ... ] \]

• Here
  • \( l_i \) is a method/field name
  • \( \zeta(x) \ b_i \) is a method where \( x \) is the self object and \( b_i \) is the body

• Operations:
  • Selection: \( o.l_i \rightarrow b_i \{ x := o \} \)
  • Override: \( o.l_i \leq \zeta(y) \ b \rightarrow [ ..., l_i = \zeta(y) \ b, ... ] \)
Simply Typed Object Calculus

• A type has the form

\[ X = [\ldots, i: Y_i, \ldots ] \quad i = 1..n \]

• The \( Y_i \) could also be \( X \), so types are potentially recursive

• The \( Y_i \) are the return values of the methods
  • All methods take a single argument of type \( X \), so the input type is omitted
The Question

• Why do we need an object calculus at all?

• There is no issue with untyped calculi
  • Object-oriented programs can be encoded in untyped lambda calculus
  • And vice-versa

• The problem is in typed calculi
Two Features Using Type Recursion

Define $A = [..., l_i: B_i, ... ]$ $i = 1..n$

$E, x_i: A \vdash b_i: B_i$ $i = 1..n$

$\overline{\text{[Object]}}$

$E \vdash [ ..., l_i = \zeta(x_i) b_i, ... ] : A$

$E \vdash a : A$ $E, x:A \vdash b: B_j$

$\overline{\text{[Override]}}$

$E \vdash a.l_j <= \zeta(x) b: A$
What’s the Problem?

• When using the lambda calculus with record types, it is difficult to model both the type recursion in object types and the recursion of override simultaneously

• Because
  • Object types depend on the types of fields, which override can change
  • Encoding objects in the lambda calculus makes it impossible to treat these separately
    • Need one uniform type system for the lambda calculus that is expressive enough to handle both the encoding of recursive types and the alterations done by override

• This turns out to be difficult and complicated
  • Which makes the resulting type systems difficult to understand and use
New Stuff
A Practical Problem

• These issues come up in all statically typed languages with object-oriented features

• When are an object’s methods defined?
• When can override be performed?

• To make both value/object recursion and override work in a statically typed language, these features are often split so that all overrides happen before any computation is done.
Solution #1: Mainstream Typed OO

• Restrict the definition of methods to a first phase before methods are typed
  • Mechanisms like inheritance, static override, restrictions on modifying superclasses, dynamic update only of fields
  • Guarantees the assembly of the object’s type is independent of program evaluation
  • Type checking happens after assembly of the methods and before the program executes

• Examples: C++, Java
Java Example

class Foo{
    public void hello() {
        System.out.println("Hello world!");
    }
}
class Bar extends Foo {
    public void hello(){
        System.out.println("Hello, user!");
    }
    public void goodbye(){
        System.out.println("Hello, user!");
    }
}

• Class Bar inherits from class Foo

• Inheritance in Java is a static property
  • A class and its parent must be explicitly named

• Method override is completely resolved at compile time
  • Even before type checking!
  • We only need the names of the classes and methods
  • The method in the subclass replaces the overridden method in the parent class

• There are type restrictions
  • A method f must have the same signature as method f in the parent class
  • Just like simply typed object calculus
  • But this can be checked after overriding is resolved
A More Practical Example

abstract class Shape {
    abstract Number calculateArea();
}

class Triangle extends Shape {
    private final double base;
    private final double height;
    ...
    double calculateArea() {
        return (base / 2) * height;
    }
}

class Square extends Shape {
    private final double side;
    ...
    double calculateArea() {
        return side * side;
    }
}

• This example shows a more typical use of override

• The base class is abstract, meaning its interface is defined but no implementation is given

• Any method in an abstract base class must have an implementation in any subclass
  • Of course the subclasses can have additional methods and fields, too

• The calculateArea method is overridden in each of the subclasses to give the appropriate implementation for the kind of shape the subclass represents

• C++ has very similar mechanisms for inheritance and override
  • Entirely static

Alex Aiken     CS 242     Lecture 8
Solution #2: Functional + OO

• Add object-oriented features to a functional language
  • Add primitive OO features to the lambda calculus

• Let the functional language do most of the work
  • The OO extensions are a thin veneer
  • Record types (or something similar) handles the typing
  • Higher-order functions give other ways to work around OO restrictions

• Every functional language has added an object system
  • Examples: OCaml, Haskell
OCaml

• Ocaml has a mix of functional, object-oriented and imperative features

• Fundamentally it is a functional language
  • Based on lambda calculus
  • OO features are implemented by translation to lambda calculus
  • Using records and record types
  • Call-by-value
let counter =
  object
    val mutable x = 0
    method get = x
    method inc = x <- x + 1
  end;

Type checker: *val counter : < get : int, inc : unit >*

Note that Ocaml is more dynamic than Java and C++
  Some new kinds of objects can be computed, not just statically defined
  But still statically typed
let counter =
  object (s)
    val mutable x = 0
    method get = s#x
    method inc = x <- x + 1
  end;
Type checker: val counter : < get : int, inc : unit >

Objects can have a self parameter, but it must be explicitly bound
OCaml

class counter =
  object (s)
    val mutable x = 0
    method get = s#x
    method inc = x <- x + 1
  end;
Type checker: class counter : < get : int, inc : unit >

Classes can also be declared at the top level. Unlike immediate objects, classes can be inherited.
let pointer = ref ...

class counter =
  object (s)
    val mutable x = 0
    method get = s#x
    method inc = x <- x + 1
    method register = pointer <- s
  end;

Type error: *Self type cannot escape its class*

Self parameters can only be used within the class in which they are bound – the can’t “escape” by being stored in global variables, for example, because then standard type checking cannot be guaranteed to give correct results. All other statically typed OO languages (Java, C++, etc.) have the same restrictions on self types.
Haskell

• A lazy functional language
  • With object-oriented and imperative features that are translated into the functional core

• Haskell takes a different approach to object-oriented features
  • The focus is on general support for overloading
Overloading: A Digression

• Two kinds of polymorphism are common in programming languages

• Subtyping
  Example: if ColorPoint extends Point, then ColorPoint can be used wherever a Point is expected

• Parametric polymorphism works for any type
  Example: cons(a,l) : `a -> list `a -> list `a

• Overloading is a set of functions with the same name
  • Only works at very specific types
  • Example: A + B
  • A,B could be integers, floats or strings
  • + is overloaded to work at just these three types
  • But three completely different implementations
Haskell Type Classes

• Type classes are a general method for overloading functions

• Consider: What is the type of the equality function $==$?

• If it is overloaded for a fixed set of types ($\text{int, bool, float, char}$) then it is inconvenient that it can’t be extended to user-defined types

• A parametric polymorphic definition doesn’t make sense
  • $==$: `$a \rightarrow \text{a} \rightarrow \text{bool}`
  • For some types, like function types, there is no sensible definition of $==$
Type Class Example

class Eq a where
    (==) :: a -> a -> Bool

Read "Any type T in the Eq type class must define a function == with signature T -> T -> Bool"

This sounds a lot like an abstract base class!
• Really very close to an abstract interface (ala Java)
Type Class Examples

class Eq a where
    (==) :: a -> a -> Bool

instance Eq Int where
    i == j = int_eq i j

class Num a where
    (*) :: a -> a -> a
    (+) :: a -> a -> a

instance Num Int where
    (*) = int_times
    (+) = int_plus
Type Class Examples

Testing if \( y \) is an element of a list:

\[
\text{member } [] \ y = \text{False} \\
\text{member } (x: \text{xs}) \ y = (x == y) \lor \text{member } \text{xs} \ y
\]

\textit{member} :: \text{list } `a \to `a \to \text{Bool} \quad \text{-- the pre-type classes type}

But \text{member} only works if == is defined on the elements of the list.

With type classes we can enforce this restriction:

\textit{member} :: \text{Eq } `a \Rightarrow \text{list } `a \to `a \to \text{Bool}
Subclasses

class Eq where
  (==) :: a -> a -> Bool

class Eq a => Num a where
  (*) :: a -> a -> a
  (+) :: a -> a -> a

``Any instance of the Eq typeclass can also be a member of the Num typeclass if it implements the additional * and + methods’’

Instances can be subclasses of multiple typeclasses

• Again, interfaces in Java, instead of single-inheritance Java classes, are the best analogy
Summary of Type Classes

• Type classes observe that inheritance/override is a form of overloading

• Unifies traditional ad hoc overloading with OO classes
  • Only two forms of polymorphism, parametric and type classes
  • And they work well together!
  • Compare with the crazy overloading rules in Java and C++

• Cost
  • Very static: Programmer must declare all type classes
  • And explicitly declare which type classes each implementation satisfies
Solution #3: OO + Functional

• Add functional features to an OO language

• Starting from a language with objects and imperative features, add
  • first-class functions
  • parametric polymorphism, if the language is typed

• Every object-oriented language has added first-class functions
  • Examples: Java and C++
Lambdas in Java

• A lambda abstraction in Java is written

  (arg) -> { function body }

• Just like lambda calculus:
  • The function is anonymous (doesn’t have a name)
  • Takes a single argument (arg in the scheme above)

• Unlike lambda calculus:
  • The function body can make use of all Java features, include objects and state
Java Lambda Example

-- print out each number in an ArrayList using forEach
numbers.forEach( (n) -> { System.out.println(n); } )

-- prints ``Hello?``
mkquestion = (s) -> s + "?";
ask = mkquestion.run(``Hello``)
Parametric Polymorphism in C++

```cpp
template <class T>
class MyNum {
private:
    T val;
public:
    MyNum(T n) : val(n) {}
    T Square() { return val * val; }
};

MyNum<int> MyNum(42);
MyNum<float> MyNum(42.0);
MyNum<Foo> MyNum (Foo); // type error!
```

- A template parameterizes a block of code on a type
  - Doesn’t have to be a class, but often is

- Type checking is done by instantiating the template and then type checking the body with the instance types substituted for the type parameters of the template
Solution #4: Dynamically Typed

- Give up on static typing
  - Go with the simplicity of dynamically typed languages

- Noticeably more popular in the OO world
  - Because static typing ends up being more complex

- Examples: Python, Javascript
  - These systems are more reminiscent of the untyped object calculus
Python Classes

class Dog:
    def bark(self):
        print("Woof!");

rover = Dog()
rover.bark()

Classes in Python have attributes (not shown) methods

All pretty conventional!

But not type checking ...
Prototypes

• Prototype-based object systems are found only in dynamically typed languages

• A prototype is a concrete object --- not a class

• In a prototype system, new objects are created by copying a prototype
  • That’s all!
  • New subtypes are defined by creating new prototypes that add behavior to a base prototype object
function Cat(name) {
    this.name = name;
    this.sound = function() { print(`meow!`); };
}

function Dog(name) {
    this.name = name;
    this.sound = function() { print(`woof!`); };
}

A = Cat(“Sleepy”);
B = Dog(“Grumpy”);
A.sound();
Meow
B.sound();
Woof
A.__proto__ = B.__proto__
A.sound()
Woof
function Cat(name) {
    this.name = name;
    this.sound = function() { print(`meow!`); }
}

function Dog(name) {
    this.name = name;
    this.sound = function() { print(`woof!`); }
}

A = Cat(“Sleepy”);
B = Dog(“Grumpy”);

-- Add a new property for cat
A.prototype.fur = “Black”;

-- change the prototype for cats
A.prototype = B.prototype
A.sound()
Woof
Prototypes, Continued

• In a prototype object system, every object has a prototype

• Objects inherit from other objects
  • With null being the initial prototype
  • Any referenced property is searched for in this prototype chain

• Since prototypes are implemented by objects, it is possible to
  • Add new properties, both fields and methods
  • Even replace the prototype with a new one
  • All dynamically

• Python has classes and added a prototype system
• Javascript has prototypes and added classes
• Since the languages are very dynamic, possible to implement any object system one wants
  • Classes and prototypes are the popular ones
Summary

• There has been a convergence of language features over the last decade
  • Mainstream languages have OO, functional, and imperative features

• There is no one best way to combine OO and functional features
  • Common cases all work in all languages
  • But there are different restrictions depending on whether the starting point is a functional language or an object-oriented language
  • Biggest divide is typed vs. untyped