Objects

CS242

Lecture 10

One More Time: The Lambda Calculus …

$$e \rightarrow x \mid \lambda x.e \mid e\ e$$

- The lambda calculus has served as a model for procedural and functional languages
  - Extremely well-studied
- Naturally people have used it to study object-oriented languages, too

Records

- Records are collections of named fields with values.

[flag = False, value = 42]

- The order of the fields is unimportant
  - [value = 42 , flag = False] is the same record

What Are Records?

- Records with a fixed set of fields are a special case of algebraic data types

```
Type Rec =
        flag: Bool
        value: Int
```

- Records are standard in many procedural languages
  - Well-studied in the lambda calculus

What is an Object?

- An object is a collection of fields and methods on those fields
- Can we use records to implement objects?
- Seems to work fine for the fields:
  [flag = False, value = 42]
- The type is analogous to the type of the corresponding algebraic data type
  [flag: Bool, value: Int]
- A *record type*
  - The type analog of a record: A collection of unordered fields, each with a type

What About Methods?

- Conceptually an object is a record of *both* fields and methods

  [flag = False, value = 42, add(i: Int): Int]

- For types we use function types

  [flag: Bool, value: Int, add: Int → Int]

But What About Self Parameters?

In every object-oriented language, methods take the self parameter either as an explicit or implicit argument

```
Class Pair {
    x : Bool
    y : Int
    first(): self.x
    second(): self.y
}

(new Pair(true,42)).second = 42
```

But What About Self Parameters?

- In every object-oriented language, methods take the self parameter either as an explicit or implicit argument
- The self parameter needs to be reflected in the type

$$X = [\ x: Bool, y: Int, first: X \rightarrow Bool, second: X \rightarrow Int]$$

- Implication: Because every method of an object o takes o as an argument, object types are recursive

Discussion

- Object types are more complex than might first be supposed
  - Inherent in the nature of the object-oriented model
- But do recursive types pose a problem?
  - Not by themselves. Recursive types are well-defined.
  - But this is a hint that things might not be so simple …

The Grand Idea

- Goal: Develop a semantics and type system(s) for objects based on the lambda calculus with records and recursive types
- Many people worked on this program over years
  - Scores, maybe hundreds, of papers
- And, surprisingly, it failed
- The conclusion was that object-orientation is best explained with a native object calculus.
  - Specifically, typed lambda calculus has trouble encoding typed object systems

Object Calculus

- Introduce in the mid-1990's by Martin Abadi and Luca Cardelli.

Untyped Object Calculus Syntax

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

  $o = [\ldots, l_i = \varsigma(x)\ b_i, \ldots]$

- Here
  - l_i is a field name
  - $\varsigma(x)\ b_i$ is a method where x is the self object and b_i is the body

- Operations:
  - Selection: $o.l_i \rightarrow bi[x := o]$
  - Override: $o.l_i <= \varsigma(y)\ b \rightarrow o$ with $l_i = \varsigma(y)\ b$

Fields vs. Methods

- We don't need to distinguish fields and methods
- Because fields are just constant methods
- Example: o = [l = ς(x) 3]
- Field lookup
  - o.l → 3[x := o] = 3
- Field update:
  - o.l <= ς(y) 4 → [l = ς(y) 4]

Recursion

- Critical to the object calculus is that the self object can appear in method bodies
  - Allows recursive behavior

- Examples
  - o = [l = ς(x) x.l]
  - o.l → x.l [x := o] = o.l →…

  - o = [l = ς(x) x]
  - o.l → x [x := o] = o

Computing with Override

- Programmatic use of override is a key feature of the object calculus

$o = [l = \varsigma(y)\ (y.l <= \varsigma(x)\ x)\ ]$

$o.l \rightarrow o.l <= \varsigma(x)\ x \rightarrow [\ l = \varsigma(x)\ x\ ]$

Examples

Recap: Implementing fields

o = [l = ς(x) 3]

o.l <= ς(y) 4 → [l = ς(y) 4]

Pairs

o = [first = ς(x) 1, second = ς(x) 2]

o.first → 1 [x := o] = 1

o.second → 2 [x := o] = 2

Examples

Predicates

o = [istrue = ς(x) true,
    setfalse = ς(x) (x.istrue <= ς(x) false),
    settrue = ς(x) (x.istrue <= ς(x) true)]

o.istrue → true

o.setfalse.istrue →
(o.istrue <= ς(x) false).istrue →
false

o.setfalse.settrue.istrue →
(o.istrue <= ς(x) false).settrue. istrue →
((o.istrue <= ς(x) false).istrue <= ς(x) true).settrue.istrue →
(o.istrue <= ς(x) true).istrue →
true

Backup Methods

o = [retrieve = ς(x) x,

backup = ς(x) x.retrieve <= ς(y) x]

Every time the backup method is called, it modifies the retrieve method to return the self object at the time the backup method was invoked.

Natural Numbers

zero = [iszero = ς(x) true,
pred = ς(x) x,
succ = ς(x) (x.iszero <= ς(y) false).pred <= ς(y) x]

Examples:

zero.iszero → true [x := …] = true

zero.succ.iszero → [iszero = ς(y) false, pred = ς(y) zero, …].iszero → false

zero.succ.pred.iszero →[iszero = ς(y) false, pred = ς(y) zero, …].pred.iszero → zero.iszero → true

Encoding Lambda Calculus with Objects

T(x) = x

$T(e_1\ e_2)$ = ($T(e_1)$.arg <= ς(y) $T(e_2)$).val

T(λx.e) = [arg = ς(x) x.arg, val = ς(x)T(e)[x := x.arg]]

The idea:

A function is represented as an object of two fields, a function argument and the function body.

When the function is defined, the argument can be anything (here it is an infinite loop). When the function is applied, the argument is overridden with the actual argument and the body is evaluated.

Note how the function argument is shared with the function body.

Example

Translation

T((λx.x) y) =

(T(λx.x).arg <= ς(z) T(y)).val =

(T(λx.x).arg <= ς(z) y).val =

([arg = ς(x) x.arg, val = ς(x)T(x)[x := x.arg]].arg <= ς(z) y).val =

([arg = ς(x) x.arg, val = ς(x)x [x := x.arg]].arg <= ς(z) y).val =

([arg = ς(x) x.arg, val = ς(x) x.arg].arg <= ς(z) y).val

Evaluation

([arg = ς(x) x.arg, val = ς(x) x.arg].arg <= ς(z) y).val →

([arg = ς(z) y, val = ς(x) x.arg]).val →

([arg = ς(z) y, val = ς(x) x.arg]).arg →

y

Encoding Objects with Lambda Calculus

- Represent objects as list of pairs
  - First component of the pair is a field/method label (an integer)
  - Second component is the value of the field/method

- Field selection
  - o.i = (o (λh.λt. if (fst h) == i then (snd h) else t) λx.x)

- Method override
  - o.i <= f = cons (i,f) o

Bottom Line

- Untyped object systems and untyped lambda calculus are easily converted one to the other
  - Equivalent in computational power

- But the same is not true for typed versions because of override
  - Recall the type X = [x: Bool, y: Int, first: X → Bool, second: X → Int]
  - Override can arbitrarily change the method, and therefore the type, of any field
    - At any point in the computation
    - Unrestricted override is not a static property

Typed Object Systems

Subtyping

- The distinctive feature of type object-oriented languages is *subtyping*

- A < B if an object of type A can be used in any context where an object of type B is required
  - Concretely, A has all the fields/methods of B

- Subtyping is a form of type polymorphism
  - Different from parametric polymorphism

Subtyping Rule for Object Calculus

$$\frac{E \vdash o: [\, l_1 : B_1, \ldots, l_n : B_n] \quad m < n}{E \vdash o: [\, l_1 : B_1, \ldots, l_m : B_m]}\ [\text{Subtyping}]$$

A Type Checking Problem

- When are an object's methods defined?
- When can override be performed?

- To have both static type checking and override, these features are often split in type object-oriented languages 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
  - But this can be checked after overriding is resolved

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

OCaml

```
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

OCaml

```
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.

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, e.g., 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++

```
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
- Closely related to polymorphic types ala Lecture 6

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 reminiscent of the untyped object calculus

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

Javascript Example

```
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
```

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 …

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

Prototypes vs Classes

- 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