Object Calculus

CS242

Lecture 7

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

What is an Object?

- Objects are collections of fields with values.

[flag = False, value = 42]

- This is an example of a *record*
    - A collection of named fields
    - Where order is unimportant
    - A standard construct in many procedural languages
    - Well-studied in the lambda calculus

What is an Object Type?

- This suggests how to give types to objects

  [flag: Bool, value: Int]

- This is a *record type*
  - The type analog of a record: A collection of unordered fields, each with a type
  - Record typing is very well-studied
  - And clearly closely related to objects and object types

What About Methods?

- Conceptually an object is a record of fields and methods

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

- For types we use function types

  [flag: Bool, value: Int, add: Int $\rightarrow$ Int]

But What About Self Parameters?

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

$$X = [\ \text{flag: Bool, value: Int, hasflag: } X \rightarrow \text{Int} \rightarrow \text{Int}\ ]$$

- 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. Infinite recursive types are perfectly well-defined.

The Program

- 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.
  - Scores, maybe hundreds, of papers

- And, surprisingly, it ultimately failed.

- The eventual conclusion was that object-orientation is best explained with a native object calculus.
  - A failure of the lambda calculus to explain objects

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

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

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

- Operations:
  - Selection: $o.l_i \rightarrow b_i\{x := o\}$
  - Override: $o.l_i \Leftarrow \varsigma(y)\ b \quad \rightarrow\ [\ \dots, l_i = \varsigma(y)\ b, \dots\ ]$

Fields vs. Methods

- We don’t need to distinguish fields and methods
- Because fields are just a special case of methods
- Example: $o = [l = \varsigma(x)\ 3]$
- Field lookup
  - $o.l \to 3[x := o] = 3$
- Field update:
  - $o.l \Leftarrow \varsigma(y)\ 4 \to [l = \varsigma(y)\ 4]$

Recursion

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

- Examples
  - $o = [l = \varsigma(x)\ x.l]$
  - $o.l \rightarrow x.l\ [x := o] = o.l \rightarrow \ldots$

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

Computing with Override

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

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

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

Encoding Lambda Calculus with Objects

$T(x) = x$

$T(e_1\ e_2) = (T(e_1).arg \Leftarrow \varsigma(y)\ T(e_2)).val$

$T(\lambda x.e) = [arg = \varsigma(x)\ x.arg,\ val = \varsigma(x) T(e)\{\ x := x.arg\ \}]$

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 argument is shared with the body through the rule for evaluation of fields.

Example

$T((\lambda x.x)\ y) =$
$(T(\lambda x.x).arg \Leftarrow \varsigma(z)\ T(y)).val =$
$(T(\lambda x.x).arg \Leftarrow \varsigma(z)\ y).val =$
$([arg = \varsigma(x)\ x.arg,\ val = \varsigma(x)T(x)\{\ x := x.arg\ \}].arg \Leftarrow \varsigma(z)\ y).val =$
$([arg = \varsigma(x)\ x.arg,\ val = \varsigma(x)x\{\ x := x.arg\ \}].arg \Leftarrow \varsigma(z)\ y).val =$
$([arg = \varsigma(x)\ x.arg,\ val = \varsigma(x)\ x.arg].arg \Leftarrow \varsigma(z)\ y).val \rightarrow$
$([arg = \varsigma(z)\ y,\ val = \varsigma(x)\ x.arg]).val \rightarrow$
$([arg = \varsigma(z)\ y,\ val = \varsigma(x)\ x.arg]).arg \rightarrow$
y

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 $\rightarrow$ true [x := …] = true

zero.succ.iszero $\rightarrow$ [iszero = ς(y) false, pred = zero, …].iszero $\rightarrow$ false

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

The (Simply) Typed Object Calculus

- A type has the form

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

- 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

Type Rules

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

$$\frac{E, x_i : A \vdash b_i : B_i \quad i = 1..n}{E \vdash [\ldots, l_i = \varsigma(x_i)\, b_i, \ldots] : A} \text{[Object]}$$

$$\frac{E \vdash o : [\ldots, l_i : B_i, \ldots]}{E \vdash o.l_j : B_j} \text{[Select]}$$

$$\frac{E \vdash a : A \quad E, x : A \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x)\, b : A} \text{[Override]}$$

Simplified Numbers

$\text{zero} = [\ \text{succ} = \varsigma(x)\ x.\text{pred} \mathrel{<=} \varsigma(y)\ x\ ]$

$\text{Nat} = [\ \text{pred}: \text{Nat},\ \text{succ}: \text{Nat}\ ]$

$$\dfrac{x: \text{Nat} \vdash x : \text{Nat} \qquad \dfrac{x: \text{Nat} \vdash x : \text{Nat} \qquad x : \text{Nat},\ y: \text{Nat} \vdash x : \text{Nat}}{x: \text{Nat} \vdash x.\text{pred} \mathrel{<=} \varsigma(y)\ x : \text{Nat}}}{\vdash [\ \text{pred} = \varsigma(x)\ x,\ \text{succ} = \varsigma(x)\ x.\text{pred} \mathrel{<=} \varsigma(y)\ x\ ]: \text{Nat}}$$

Adding Subtyping

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

Bottom line: We can add subtyping, and it works as expected.

Discussion

- Back to 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 Kinds of Recursion

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

$$\frac{E, x_i : A \vdash b_i : B_i \qquad i = 1..n}{E \vdash [\ \ldots, l_i = \varsigma(x_i)\ b_i, \ldots\ ] : A}\ \text{[Object]}$$

$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x)\ b : A}\ \text{[Override]}$$

What’s the Problem?

- When using record calculi, it is easy to model either the recursion of object values, or to model the recursion of override

- But not both
  - Fixing one breaks the other
  - Because in encodings in the lambda calculus, these two operations require two separate recursive bindings at different times

A Practical Problem

- This issue comes up in all object-oriented languages

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

- To make both value/object recursion and override work in a statically type language, they are often split so that all overrides happen before any computation is done.

Approach #1: Mainstream Typed OO

- Restrict the definition of methods to a phase before the program is run
  - Mechanisms like inheritance, static override, restrictions on modifying superclass, 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

Approach #2: Functional + OO

- Add object-oriented features to a functional (lambda calculus-based) language

- 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

- Examples: Haskell, OCaml

Approach #3: OO + Functional

- Add functional features to an OO language
- And have a *much* more complex type system
  - More dynamic, too, with fewer restrictions on how classes are assembled from methods
- Example: Scala

Approach #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
  - E.g., prototype-based classes

Summary

- Even the simple settings for object-oriented languages raise issues not found in the lambda calculus
- Key insight is that method override is difficult to encode in systems other than a native object calculus
    - At least in conjunction with recursive object types
- This issue leads to a wide variety of approaches to type systems in object-oriented languages
    - And explains popular restrictions, like all overrides before any evaluation
    - Or simply going with a dynamically typed language