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

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

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

- 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.
  - But this is a hint that things might not be so simple …

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 over years
  - 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 typed lambda calculus to explain typed 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

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

- 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: $o.l_i \rightarrow b_i\{x := o\}$
  - Override: $o.l_i <= \varsigma(y)\ b \rightarrow o \text{ with } l_i = \varsigma(y)\ b$

Fields vs. Methods

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

Recursion

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

- Examples
  - o = [l = 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.l [x := o] = 1

o.second → 2.l [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.l → 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

Encoding Lambda Calculus with Objects

$T(x) = x$

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

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

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

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

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 = zero, …].iszero → false

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

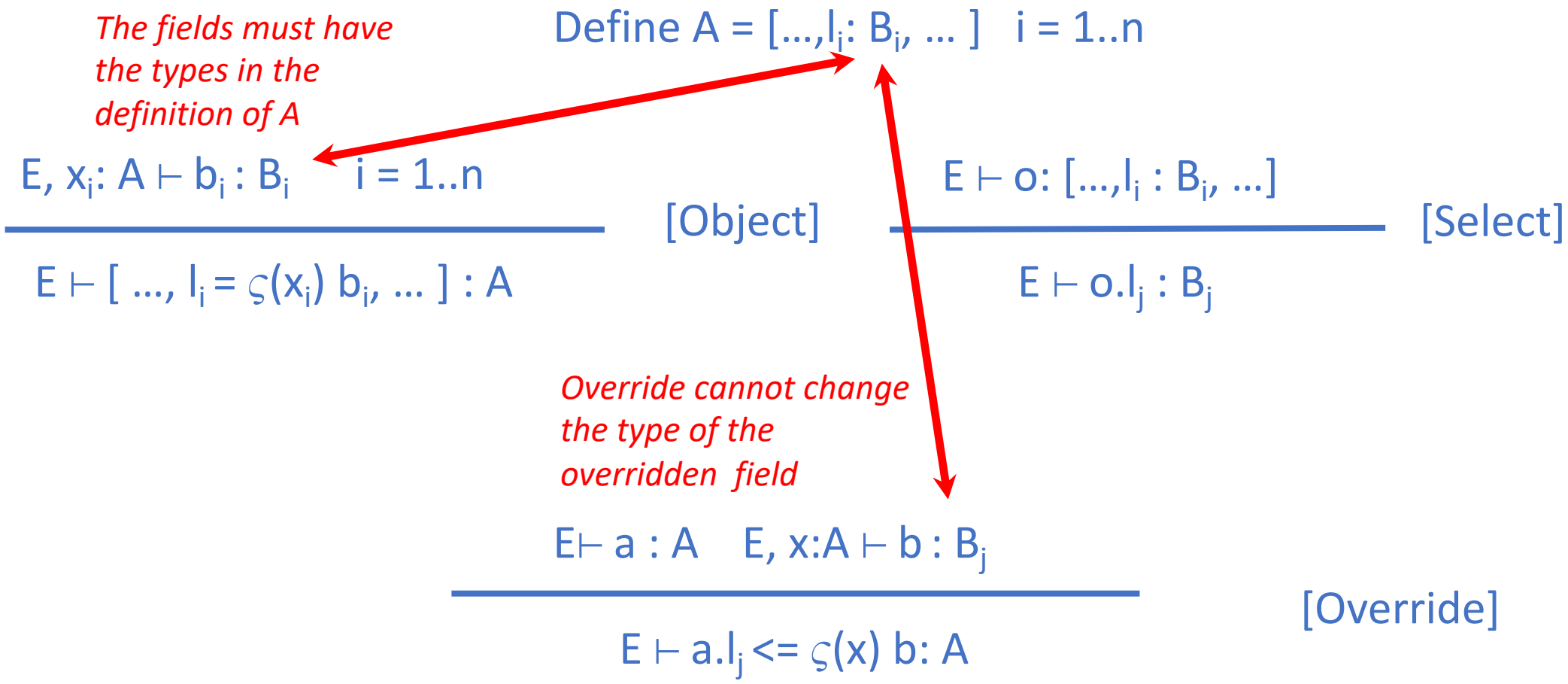
The (Simply) Typed Object Calculus

- 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
  - Typical of OO type systems!
    - The recursive nature of the types is implicit

Type Rules


The fields must have the types in the definition of A
Define A = [...,l_i: B_i, ...] i = 1..n
E, x_i: A ⊢ b_i : B_i i = 1..n
E ⊢ [..., l_i = ς(x_i) b_i, ...] : A
[Object]
E ⊢ o: [...,l_i : B_i, ...]
E ⊢ o.l_j : B_j
[Select]
Override cannot change the type of the overridden field
E⊢ a : A E, x:A ⊢ b : B_j
E ⊢ a.l_j <= ς(x) b: A
[Override]

Simplified Numbers

zero = [succ = ς(x) x.pred <= ς(y) x]

Nat = [pred: Nat, succ: Nat]

$$\frac{x\text{: Nat} \vdash x : \text{Nat} \qquad x : \text{Nat}, y\text{: Nat} \vdash x : \text{Nat}}{x\text{: Nat} \vdash x.\text{pred} <= \varsigma(y)\ x : \text{Nat}} \qquad x\text{: Nat} \vdash x : \text{Nat}$$

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

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 Features Using Type Recursion

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

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

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

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 type recursion of override simultaneously

- Because
  - Object types depend on the types of fields
  - Override can replace methods with new methods that use fields in new ways
  - No one has found a translation into lambda calculus that separates these two issues
  - 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

- All known approaches have resulted in complex type systems
  - Which makes the resulting type systems difficult to understand and use

Next Time

- The impact of this discovery has led to a split in the typing of object oriented languages
- Or rather, it has failed to resolve the existing splits
- Next time we'll look at the various solutions for object systems in practice