Modeling and Understanding Object-Oriented Programming

Cunning Plan: Focus On Objects
- A Calculus For OO
- Operational Semantics
- Type System
- Expressive Power
- Encoding OO Features

The Need for a Calculus
- There are many OO languages with many combinations of features
- We would like to study these features formally in the context of some primitive language
  - Small, essential, flexible
- We want a “λ-calculus” or “IMP” for objects

Why Not Use λ-Calculus for OO?
- We could define some aspects of OO languages using λ-calculus
  - e.g., the operational semantics by means of a translation to λ-calculus
- But then the notion of object be secondary
  - Functions would still be first-class citizens
- Some typing considerations of OO languages are hard to express in λ-calculus
  - i.e., object-orientation is not simply “syntactic sugar”

Object Calculi Summary
- As in λ-calculi we have
  - operational semantics
  - denotational semantics
  - type systems
  - type inference algorithms
  - guidance for language design
- We will actually present a family of calculi
  - typed and untyped
  - first-order and higher-order type systems
- We start with an untyped calculus

An Untyped Object Calculus
- An object is a collection of methods
  - Their order does not matter
- Each method has
  - A bound variable for “self” (denoting the host object)
  - A body that produces a result
- The only operations on objects are:
  - Method invocations
  - Method update
Untyped Object Calculus Syntax

- Syntax:
  - variables
    - $a, b ::= x$ - variables
    - $[mi = \varsigma(x) bi]$ - object constructor
      - $\varsigma$ is a variant of Greek letter $\sigma$
      - $x$ is the local name for “self”
  - method invocation
    - $a.m$ - method invocation
      - no arguments (just the self)
  - method update
    - $a.m ← \varsigma(x)b$ - method update
      - this is an expression!
      - the result is a copy of the object
      - with one method changed
- This is called the untyped $\varsigma$-calculus (Abadi & Cardelli)

First Examples

- An object $o$ with two methods $m_1$ and $m_2$
  - $m_1$ returns an empty object
  - $m_2$ invokes $m_1$ through self
    - $o = [m_1 = \varsigma(x) [], \ m_2 = \varsigma(x). m_1 ]$
- A bit cell with three methods: value, set and reset
  - value returns the value of the bit (0 initially)
  - set sets the value to 1, reset sets the value to 0
  - models state without $\lambda$/IMP (objects are primary)
    - $b = [value = \varsigma(x). 0,$
      - $set = \varsigma(x). x.value ← \varsigma(y). 1,$
      - $reset = \varsigma(x). x.value ← \varsigma(y). 0 ]$

Operational Semantics

- $a → b$ means that $a$ reduces in one step to $b$
- The rules are: (let $o$ be the object $[mi = \varsigma(x). bi]$)
  - $o.m_i → [o/x] b_i$
  - $o.mk ← \varsigma(y). b → [mk = \varsigma(y). b, \ mi = \varsigma(x). bi]$  
    - $i ∈ \{1, ..., n\} - \{k\}$
- We are dealing with a calculus of objects
- This is a deterministic semantics (has the Church-Rosser or “diamond” property)

Expressiveness

- A calculus based only on methods with “self”
  - How expressive is this language? Let’s see.
  - Can we encode languages with fields? Yes.
  - Can we encode classes and subclassing? Hmm.
  - Can we encode $\lambda$-calculus? Hmm.
- Encoding fields
  - Fields are methods that do not use self
  - Field access “$o.f$” is translated directly
    - to method invocation “$o.f$”
  - Field update “$o.f ← e$” is translated to “$o.f ← \varsigma(x). e$”
  - We will drop the $\varsigma(x)$ from field definitions and updates

As Expressive As $\lambda$

- Encoding functions
  - A function is an object with two methods
    - $arg$ - the actual value of the argument
    - $val$ - the body of the function
  - A function call updates “arg” and invokes “val”
- A conversion from $\lambda$-calculus expressions
  - $\lambda x. e = x.arg$ (read the actual argument)
  - $\vec{e}_1 e_2 = (e_1.arg ← \varsigma(y) e_2).val$
  - $\lambda \vec{e}. e = [arg = \varsigma(y) y.arg, val = \varsigma(x). e ]$
  - The initial value of the argument is undefined
  - From now on we use $\lambda$ notation in addition to $\varsigma$

$\lambda$-calculus into $\varsigma$-calculus

- Consider the conversion of $(\lambda x. x) 5$
  - Let $o = [ arg = \varsigma(y) 5, arg = \varsigma(x). x.arg]$
  - $(\lambda x. x) 5 = (o.arg ← \varsigma(y) 5).val$
- Consider now the evaluation of this latter $\varsigma$-term
  - Let $o' = [ arg = \varsigma(y) 5, val = \varsigma(x). x.arg]$
  - $(o.arg ← \varsigma(y) 5).val → o'.val = [arg = \varsigma(y) 5, val = \varsigma(x). x.arg].val → x.arg[o'/x] = o'.arg → 5[o'/y] = 5$
Encoding Classes

• A **class** is just an object with a “new” method, for generating new objects
  - A repository of code for the methods of the generated objects (so that generated objects do not carry the methods with them)
• Example: for generating o = [m₁ = \(\varsigma(x)\) b₁]
  c = [new = \(\varsigma(z)\) [m₁ = \(\varsigma(x)\) z.m₁ x],
  \(m₁ = \varsigma(\text{self})\) \(\lambda x.\) b₁]
  - The object can also carry “updateable” methods
  - Note that the \(m₁\) in c are fields (don’t use \text{self}"

Class Encoding Example

• A class of bit cells
  BitClass = [ new = \(\varsigma(z)\). \(\text{val} = \varsigma(x) 0,\)
  set = \(\varsigma(x)\) \(\text{BitClass.set} x,\)
  reset = \(\varsigma(x)\) \(\text{BitClass.reset} x\) ],
  \(\text{set} = \varsigma(z) \lambda x. x.\text{val} \leftarrow \varsigma(y) 1,\)
  \(\text{reset} = \varsigma(z) \lambda x. x.\text{val} \leftarrow \varsigma(y) 0\) ]
• Example:
  BitClass.new → [ \(\text{val} = \varsigma(x) 0,\)
  \(\text{set} = \varsigma(x)\) \(\text{BitClass.set} x,\)
  \(\text{reset} = \varsigma(x)\) \(\text{BitClass.reset} x\) ]
  - The new object carries with it its identity
  - The indirection through BitClass expresses the **dynamic dispatch** through the BitClass method table

Inheritance and Subclassing

• **Inheritance** involves re-using method bodies
  FlipBitClass = [ new = \(\varsigma(z)\) \(\text{BitClass.new}\). \(\text{flip} \leftarrow \varsigma(x)\) \(\text{z.flip} x,\)
  \(\text{flip} = \varsigma(z) \lambda x. x.\text{val} \leftarrow \text{not} (x.\text{val})\) ]
• Example:
  FlipBitClass.new → [ \(\text{val} = \varsigma(x) 0,\)
  \(\text{set} = \varsigma(x)\) \(\text{BitClass.set} x,\)
  \(\text{reset} = \varsigma(x)\) \(\text{BitClass.reset} x,\)
  \(\text{flip} = \varsigma(x)\) \(\text{FlipBitClass.flip} x\) ]
  - We can model **method overriding** in a similar way

Object Types

• The previous calculus was **untyped**
• Can write invocations of nonexistent methods
  [foo = \(\varsigma(x)\) ...].bogus
• We want a type system that guarantees that well-typed expressions only invoke existing methods
• First attempt:
  - An object’s type specifies the methods it has available:
    \(A ::= [m₁, m₂, ..., mₙ]\)
  - Not good enough:
    - If \(o : [m₁, ...]\) then we still don’t know if \(o.m\) is safe
    - We also need the type of the result of a method

First-Order Object Types. Subtyping

• Second attempt:
  \(A ::= [m₁ : A₁]\)
  - Specify the available methods and their result types
  - Wherever an object is usable another with more methods should also be usable
    - This can be expressed using **width** subtyping:
    \[
    \begin{array}{c}
    \frac{A < B \quad B < C}{A < C} \\
    \frac{m₁ : A₁, \ldots, mₙ : Aₙ}{[m₁ : A₁, \ldots, mₙ : Aₙ] < [m₁ : A₁, \ldots, mₙ : Aₙ]} \end{array}
    \]

Typing Rules

• **making an object**
  \(\Gamma, x : A ⊢ b_i : A_i\)
• **invoking a method**
  \(\Gamma ⊢ b : A \quad m_i : A_i ∈ A\)
  \(\Gamma ⊢ [m_i = \varsigma(x : A) \cdot b_i] : A\)
• **updating a method**
  \(\Gamma ⊢ m_i : A_i \in A\)
  \(\Gamma, x : A ⊢ b' : A_i\)
  \(\Gamma ⊢ \text{b.m_i} ← \varsigma(x) b' : A\)
Type System Results

• Theorem (Minimum types)
  - If $\Gamma \vdash a : A$ then there exists $B$ such that for any $A'$ such that $\Gamma \vdash a : A'$ we have $B < A'$
  - If an expression has a type $A$ then it has a minimum (most precise) type $B$

• Theorem (Subject reduction)
  - If $\emptyset \vdash a : A$ and $a \to v$ then $\emptyset \vdash v : A$
  - Type preservation. Evaluating a well-typed expression yields a value of the same type.

Type Examples

• Consider that old BitCell object
  
  $o = \{ \text{value} = \varsigma(x).0, \text{set} = \varsigma(x).x.\text{value} \leftarrow \varsigma(y).1, \text{reset} = \varsigma(x).x.\text{value} \leftarrow \varsigma(y).0 \}$

  • An appropriate type for it would be
    
    $\text{BitType} = [\text{value} : \text{int}, \text{set} : \text{BitType}, \text{reset} : \text{BitType}]$

    - Note that this is a recursive type
    - Consider part of the derivation that $o : \text{BitType}$ (for set)

Unsoundness of Covariance

• Object types are invariant (not co/contravariant)

• Example of covariance being unsafe:
  - Let $U = []$ and $L = [m : U]$
  - By our rules $L < U$
  - Let $P = [x : U, f : U]$ and $Q = [x : L, f : U]$
  - Assume we (mistakenly) say that $Q < P$ (hoping for covariance in the type of $x$)
  - Consider the expression:
    
    $q : Q = [x = [m = []], f = \varsigma(s : Q).s.x.m ]$

    - Then $q : P$ (by subsumption with $Q < P$)
    - Hence $q.x \leftarrow [] : P$
    - This yields the object $[x = [], f = \varsigma(s : Q).s.x.m ]$
    - Hence $(q.x \leftarrow []).f : U$ yet $(q.x \leftarrow []).f$ fails!

Covariance Would Be Nice Though

• Recall the type of bit cells
  
  $\text{BitType} = [\text{value} : \text{int}, \text{set} : \text{BitType}, \text{reset} : \text{BitType}]$

• Consider the type of flipable bit cells
  
  $\text{FlipBitType} = [\text{value} : \text{int}, \text{set} : \text{FlipBitType}, \text{reset} : \text{FlipBitType}, \text{flip} : \text{FlipBitType}]$

  - We would expect that $\text{FlipBitType} < \text{BitType}$
  - Does not work because object types are invariant
  - We need covariance + subtyping of recursive types
    - Several ways to fix this

Variance Annotations

• Covariance fails if the method can be updated
  - If we never update set, reset or flip we could allow covariance

• We annotate each method in an object type with a variance:
  + means read-only. Method invocation but not update
  - means write-only. Method update but not invocation
  0 means read-write. Allows both update and invocation

• We must change the typing rules to check annotations
  - And we can relax the subtyping rules

Subtyping with Variance Annotations

• Invariant subtyping (Read-Write)
  
  $[\ldots \text{m}_0 : B \ldots] < [\ldots \text{m}_0 : B' \ldots]$ if $B = B'$

• Covariant subtyping (Read-only)
  
  $[\ldots \text{m}_0 : B \ldots] < [\ldots \text{m}_0 : B' \ldots]$ if $B < B'$

• Contravariant subtyping (Write-only)
  
  $[\ldots \text{m}_0 : B \ldots] < [\ldots \text{m}_0 : B' \ldots]$ if $B' < B$

• In some languages these annotations are implicit
  - e.g., only fields can be updated
Classes, Types and Variance

• Recall the type of bit cells
  \( \text{BitType} = [\text{value}^0 : \text{int}, \text{set}^+ : \text{BitType}, \text{reset}^+ : \text{BitType}] \)

• Consider the type of flipable bit cells
  \( \text{FlipBitType} = [\text{value}^0 : \text{int}, \text{set}^+ : \text{FlipBitType}, \text{reset}^+ : \text{FlipBitType}, \text{flip}^+ : \text{FlipBitType}] \)

• Now we have \( \text{FlipBitType} < \text{BitType} \)

  - Recall the subtyping rule for recursive types
  \[
  \frac{\tau < \sigma}{\mu \text{FlipBitType}.\tau < \mu \text{BitType}.\sigma}
  \]

Higher-Order Object Types

• We can define \textit{bounded polymorphism}

• Example: we want to add a method to \text{BitType} that can copy the bit value of self to another object
  \[
  \text{lendVal} = \lambda (z) (x : t < \text{BitType}). x.\text{val} \leftarrow z.\text{val}
  \]
  - Can be applied to a \text{BitType} or a subtype
  - Returns something of the same type as the input
  - Can infer that “\( z.\text{lendVal} y : \text{FlipBitType} \)” if “\( y : \text{FlipBitType} \)”

• We can add \textit{bounded existential types}
  
  - Ex: abstract type with interface “make” and “and”
    \[
    \text{Bits} = \exists t < \text{BitType}. \{\text{make} : \text{nat} \rightarrow t, \text{and} : t \rightarrow t \rightarrow t\}
    \]
  - We only know the representation type \( t < \text{BitType} \)

Classes and Types

• Let \( A = [m_1 : B_j] \) be an object type

• Let \( \text{Class}(A) \) be the type of classes for objects of type \( A \)
  \[
  \text{Class}(A) = [\text{new} : A, m_1 : A \rightarrow B_j]
  \]
  - A class has a generator and the body for the methods

• Types are distinct from classes
  - A class is a “stamp” for creating objects
  - Many classes can create objects of the same type
  - Some languages take the view that two objects have the same type only if they are created from the same class
    - With this restriction, types are classes
  - In Java both classes and interfaces act as types

Conclusions

• Object calculi are both \textit{simple and expressive}

• Simple: just \textit{method update} and \textit{method invocation}

• Functions vs. objects
  - Functions can be translated into objects
  - Objects can also be translated into functions
    - But we need sophisticated type systems
    - A complicated translation

• Classes vs. objects
  - Class-based features can be encoded with objects: subclassing, inheritance, overriding

Homework

• Good luck with your project presentations!
• Have a lovely summer.